

**Automatic Program Analysis
in a
Prolog Intelligent Teaching System**

Chee-Kit Looi

**PhD
University of Edinburgh
1988**



Abstract

This thesis proposes an approach to the analysis of novices' Prolog programs in a Prolog Intelligent Teaching System (**PITS**) that detects errors in them and proposes the corrections necessary to fix them. In order to debug Prolog programs for tutoring purposes, several different kinds of knowledge are required. Knowledge of the programming task is important. For any task, there can be different strategies or algorithms, and for each algorithm, there can be different ways of implementing it. To address this requirement, we propose the use of an *algorithm-based* approach in inferring what a novice program is intended to do and how it was intended to work, and identifying errors in these intentions and their realisations. Knowledge of the programming language is also important. Such knowledge is used to analyse the intended behaviour and implementation of the program as against its actual behaviour and implementation. To address this requirement, we propose the application of several *sources of program expertise* to analyse a Prolog program and reason about mode and type information, dataflow analysis, misspelling checking, program transformations and detect the use of Prolog programming techniques. A program analysis methodology needs to integrate reasoning with all these different kinds of knowledge.

There are two main kinds of program debugging methodologies – *static analysis* that examines the code, and *dynamic analysis* that examines the running of the code on specific examples. To harness the advantages of both static and dynamic analysis for debugging Prolog programs, we argue that a combination of debugging approaches coupled with several sources of expertise is important for debugging Prolog programs for tutoring purposes. This contrasts with other program analysis systems which tend to use just one main debugging methodology. A bug diagnosis system based on this methodology has been implemented in a computer program called **APROPOS2**. It uses a *heuristic best-first* search of the program decomposition space to parse a Prolog program into a hierarchical structure of predicate definitions, clauses, subgoals, arguments and terms. Information derived from multiple sources of program analyses is used to guide this search. APROPOS2 relies on this approach to analyse programs written by novices, diagnose each program's non-syntactic bugs and describe the bugs to the students. We address a small range of list and number manipulation tasks that are typically encountered in a Prolog programming curriculum. We carried out a *formative survey* of APROPOS2 by testing it on a corpus of students' programs for four programming tasks. The intention was to gather data on how APROPOS2 works on real students' programs which can be used to further improve APROPOS2. From our empirical experiments, we judge APROPOS2 to be feasible and useful for debugging actual students' Prolog programs.

Some of the material reported in this thesis will be published elsewhere :

- A discussion of representations of task and programming knowledge in Intelligent Tutoring Systems for programming, described in Section 2.3, in the Proceedings of the Fifth International Conference on Technology and Education, Edinburgh, Scotland, 1988;
- A description of the program analysis approach of APROPOS2, described in Chapter 3, in the Proceedings of the International Conference on Intelligent Tutoring Systems, Montreal, Canada, 1988; and in the Proceedings of the Eighth European Conference on Artificial Intelligence, Munich, West Germany, 1988.

I declare that this thesis has been composed by myself and that the work described in it is my own :

Chee-Kit Looi

Acknowledgements

I would like to express my deepest thanks to Peter Ross for his supervision of this work. Key ideas in this work can be traced to his original PITS paper and I am indeed glad to be able to draw upon his experience with teaching Prolog for many years in the department. I would like to thank Helen Pain for her invaluable advice at various points during the development of this work, and for organising the Prolog workshop in Abingdon in February 1987 and inviting me to it. I am also indebted to Jim Howe for offering me a chance to do research here in this department and for his encouragement and support especially in the initial stages of this work. Peter, Helen, Paul Brna, Chris Mellish and James Kwa read drafts of this thesis and provided apropos comments. Thanks go also to John Jones, Liam Lynch, Chris Mellish, Maarten van Someren and Gordon McCalla who read various drafts of my work and offered me helpful suggestions. Paul Brna, John Hallam, Robin Cooper, and students of the AI-2 classes of 1986/87 and 1987/88 and the MSc class of 1986/87 all helped me in my empirical evaluation studies. I would also like to thank present and former postgraduate students who provided help in many ways and made my stay in Edinburgh less formidable : Li-Dong Cai, Richard Caley, Alison Cawsey, Roberto Desimone, Ajay Gupta, David Hintze, James Kwa, John Lewis, Bing Liu, Max Longley, Gary Roberts, Richard White, Colin Williams, and Karen Valley.

This work has been supported by an University of Edinburgh studentship and a scholarship from the Tan Kah Kee Scholarship Foundation of the Chinese Chamber of Commerce, Singapore.

Table of Contents

1. Introduction.....	1
1.1 Why Automatic Analysis of Novices' Prolog Programs	
1.2 Why Shapiro's Bug Diagnosis System is Inadequate	
1.3 Why We Need Multiple Sources of Program Analysis	
1.4 Why We Need Knowledge of Algorithms	
1.5 A Combination of Debugging Approaches for APROPOS2	
1.6 An Example of APROPOS2's Output	
1.7 A Formative Evaluation of APROPOS2	
1.8 The Contributions of this Thesis	
1.9 Preliminaries for Prolog	
1.10 Structure of this Thesis	
2. The Problem of Debugging	16
2.1 The General Problem of Program Debugging	
2.2 The Problem of Debugging Prolog Programs	
2.3 Representations of Task and Programming Knowledge	
2.3.1 Prolog Programming Techniques	
2.3.2 Programming Plans	
2.3.3 Cliches	
2.3.4 Production Rules	
2.3.5 Algorithms	
2.4 Classifications of Prolog Bugs	
2.5 Alternative Automated Debugging Approaches	
2.5.1 Dynamic Analysis	
2.5.2 Code Analysis	
2.5.2.1 Structure-Based Analysis	
2.5.2.2 Symbolic Evaluation	
2.5.2.3 Program Verification	
2.5.3 What Debugging Approach for Prolog	
3. An Overview of APROPOS2	46
3.1 Multiple Sources of Program Analysis	
3.1.1 Mode Analysis	
3.1.1.1 Inferring Modes	
3.1.1.2 Uses of Mode Information	
3.1.2 Dataflow Analysis	
3.1.3 Type Analysis	
3.1.3.1 Inferring Types	
3.1.3.2 Uses of Type Information	
3.1.4 Misspelling Analysis	
3.1.5 Program Transformations	
3.1.6 Detecting Programming Techniques	
3.2 The Task-Specific Debugging Capabilities of APROPOS2	
3.2.1 The Algorithm Level	
3.2.2 The Predicate Definition Level	
3.2.3 The Implementation Level	
3.2.4 Top-Down Analysis to provide Context	
3.3 A Quick Case Study : reverse/2	
3.4 Summary	

4. A Frame Representation for Prolog Predicate Definitions.....	69
4.1 P-frame Representation	
4.1.1 Reference P-frames	
4.1.2 Student P-frames	
4.1.3 Matching Student P-frames to Reference P-frames	
4.2 Algorithms for Tasks	
4.3 How Knowledge in P-frames is Derived	
5. Algorithm Recognition, Bug Detection and Bug Correction.....	80
5.1 The Search Space	
5.2 Best-First Search	
5.3 Measuring P-frames' Mismatch	
5.4 Matching Permutations of Predicate Arguments	
5.5 The Code-Critique Phase	
5.6 Dynamic Analysis of Programs	
5.7 A Detailed Case Study : sorting/2	
5.7.1 Overview of Case Study	
5.7.2 Details of Case Study	
5.8 Summary	
6. Development of APROPOS.....	110
6.1 Chronological Development of APROPOS	
6.2 Feature Extraction in APROPOS1	
6.3 Empirical Evaluation of APROPOS1	
6.3.1 Set-Up of Evaluation of APROPOS1	
6.3.2 Discussion of APROPOS1's Program Analyses	
6.4 Analysis-by-Synthesis in APROPOS-AS	
6.5 Empirical Evaluation of APROPOS-AS	
6.6 Summary	
7. Empirical Evaluation of APROPOS2	130
7.1 Set-Up of the Evaluation	
7.2 The reverse/2 task	
7.3 The replace.elements/4 task	
7.4 The atom.count/2 task	
7.5 The fringe.length/2 task	
7.6 The sorting/2 task	
7.7 An Independent Judgement of APROPOS2's Analysis	
7.8 Discussion of Observations Drawn	
7.9 Intent of Evaluation Study	
8. Further Directions for APROPOS2	163
8.1 Analysing APROPOS2's Heuristic Code-Matching Process	
8.2 Exploring a Generative Theory of Programming	
8.3 Developing More Abstract DataFlow Representations	
8.4 Further Reasoning with Modes	
8.5 Utilising More Rigorous Sources of Program Analysis	
9. Further Directions for a PITS.....	171
9.1 The Role of a PITS	
9.2 The Role of APROPOS2 in a PITS	
9.3 Future Work for a PITS	
10. Conclusion.....	178

References	180
I.1. Brief Descriptions of Algorithms for reverse/2	186
I.2. Brief Descriptions of Algorithms for replace.elements/4	187
I.3. Brief Descriptions of Algorithms for atom.count/2	189
I.4. Brief Descriptions of Algorithms for fringe.length/2	191
I.5. Brief Descriptions of Algorithms for sorting/2	194
II. A Session Listing of APROPOS1's Analyses of a Student's reverse/2 programs	196
III. A Session Listing of APROPOS-AS's Analyses of a Student's reverse/2 programs	201
IV. Examples of APROPOS2's Analyses of Students' reverse/2 programs	206
V. An Independent Judgement of APROPOS2's Analyses of reverse/2 programs	210

List of Figures

Figure 1-1 : A student's incorrect solution for sorting/2	3
Figure 1-2 : Error diagnosis using a Shapiro's bug diagnosis system	4
Figure 1-3 : An overview of debugging in APROPOS2	9
Figure 1-4 : APROPOS2's analysis of a sorting/2 program	10
Figure 3-1 : Multiple program analyses in APROPOS2	47
Figure 3-2 : A buggy solution to reverse/2	63
Figure 3-3 : Task information for naive reverse/2	65
Figure 3-4 : APROPOS2's analysis of a student's naive reverse/2 program	67
Figure 3-5 : APROPOS2's further analysis of student's naive reverse/2 program	68
Figure 4-1 : Reference P-frame for naive algorithm for doing reverse/2	71
Figure 4-2 : Reference P-frame for inverse naive algorithm for doing reverse/2	72
Figure 4-3 : Student's program and P-frames for inverse naive reverse/2	74
Figure 5-1 : Algorithm recognition in APROPOS2	83
Figure 5-2 : Algorithm for APROPOS2's best-first search	84
Figure 5-3 : Dynamic analysis in APROPOS2 to find equivalence of predicates	97
Figure 5-4 : A buggy solution to sorting/2	99
Figure 5-5 : APROPOS2's analysis of sorting/2 program in Figure 6-4	101
Figure 5-6 : Best-first search for sorting/2 program in Figure 6-4	103
Figure 5-7 : Match of student insert/3 P-frame with reference declarative-insert/3 P-frame with argument permutation [1, 2, 3]	106
Figure 5-8 : Match of student insert/3 P-frame with reference declarative-insert/3 P-frame with argument permutation [2, 1, 3]	107
Figure 6-1 : Chronological development of various APROPOS systems	111
Figure 6-2 : Main task information for naive reverse/2	113
Figure 6-3 : An example of APROPOS1's analysis of a student's reverse/2	115
Figure 6-4 : APROPOS1's inability to suggest correction to a buggy clause	118
Figure 6-5 : APROPOS1's too terse explanation capability	119
Figure 6-6 : An example of APROPOS-AS's analysis	126
Figure 7-1 : Two best algorithm matches for a reverse/2 program by APROPOS2	135
Figure 7-2 : A correct reverse/2 program judged as incorrect by APROPOS2	136
Figure 7-3 : A correct algorithm match of a reverse/2 program by APROPOS2	137
Figure 7-4 : Another correct reverse/2 program judged as incorrect by APROPOS2	138
Figure 7-5 : An incorrect reverse/2 program judged as correct by APROPOS2	139
Figure 7-6 : An incorrect algorithm match of a reverse/2 program by APROPOS2	140
Figure 7-7 : A correct algorithm match of a reverse/2 program by APROPOS2	141
Figure 7-8 : Some student reverse/2 programs whose algorithms were unclear	142
Figure 7-9 : APROPOS2's correct algorithm identification of a replace_elements/4 program	145
Figure 7-10 : APROPOS2's incorrect algorithm identification of a replace_elements/4 program	146
Figure 7-11 : APROPOS2's analysis of a student's atom_count/2 program	147
Figure 7-12 : APROPOS2's incorrect algorithm identification of an atom_count/2 program	150
Figure 7-13 : APROPOS2's analysis of a student's fringe_length/2 program	152
Figure 7-14 : APROPOS2's incomplete analysis of a student's fringe_length/2 program	153

Figure 7-15 : APROPOS2's failure to recognise the algorithm in a student's fringe_length/2 program.....	154
Figure 7-16 : A student's first attempt at sorting/2	155
Figure 7-17 : APROPOS2's analysis of a sorting/2 program	156
Figure 7-18 : reverse/2 programs written by students with buggy models of Prolog	158
Figure 7-19 : reverse/2 program written by student whose algorithm is unclear	158
Figure 9-1 : PITS architecture using a blackboard system	173
Figure 9-2 : Role of APROPOS2 in a PITS	175

List of Tables

Table 7-1 : Results of APROPOS2's analyses of correctness of students'	
reverse/2 programs.....	133
Table 7-2 : Results of APROPOS2's analyses of algorithm recognition of students'	
reverse/2 programs.....	133
Table 7-3 : Results of APROPOS2's analyses of bug detection and correction of students'	
reverse/2 programs.....	133
Table 7-4 : Results of APROPOS2's analyses of students'	
replace_elements/4 programs	143
Table 7-5 : Results of APROPOS2's analyses of students'	
atom_count/2 programs	147
Table 7-6 : Results of APROPOS2's analyses of students'	
fringe_length/2 programs	151

apropos *adj* both relevant and opportune
Longman Family Dictionary,
Chancellor Press, 1984

Chapter 1. Introduction

This thesis proposes an approach to the analysis of novices' Prolog programs in a Prolog Intelligent Teaching System (PITS) that detects errors in them and proposes the corrections necessary to fix them. In order to debug Prolog programs for tutoring purposes, several different kinds of knowledge are required. Knowledge of the programming task is important. For any task, there can be different strategies or algorithms, and for each algorithm, there can be different ways of implementing it. Knowledge of the programming language is also important. Such knowledge is used to analyse the intended behaviour and implementation of the program as against its actual behaviour and implementation. Knowledge of prototypical bugs that novices are observed to make is used to help detect the occurrence of such bugs in students' programs and explain them in a proper context. A debugging methodology needs to integrate reasoning with all these different kinds of knowledge.

We propose the use of an algorithm-based approach in inferring what a novice program is intended to do and how it was intended to work, and identifying errors in these intentions and their realisations. In this thesis, we will show that the effective analysis of Prolog programs requires the use of several types of program analyses which may be combined for greater power. A bug diagnosis system based on this methodology has been implemented in a computer program called **APROPOS2**. APROPOS stands for Analyser of PROlog Programs Of Students and is implemented in Edinburgh Prolog [Hutchings et al 86]. It uses a heuristic best-first search of the program decomposition space to parse a Prolog program into a hierarchical structure of predicate definitions, clauses, subgoals, arguments and terms. Information derived from multiple sources of program analyses are used to guide this search. APROPOS2 relies on this approach to analyse programs written by novices, diagnose each program's non-syntactic bugs and describe the bugs to the students.

Our research is concerned with the diagnosis of bugs in student Prolog programs. An important premise of this research is that knowledge of the location and type of all bugs and the corrections necessary to fix the bugs will greatly facilitate the inference of misconceptions. Joni, Soloway, Goldman and Ehrlich [Joni, 83] distinguish bugs from misconceptions : "a *bug* is an error in a computer program, while a *misconception* is some conceptualisation in the student's mind that can lead to a program bug."

We address a small range of list and number manipulation tasks that are typically encountered in a Prolog programming curriculum and show how an automated debugger can analyse students' programs for these tasks to find out if they meet the task specifications, detect the bugs in them, suggest the likely misconceptions involved, and propose corrections

to the bugs. Our approach to debugging Prolog programs for tutoring purposes is not prejudiced towards any particular way of viewing the action of a PITS or any way of viewing how Prolog should be taught to students.

1.1 Why Automatic Analyses of Novices' Prolog Programs

Empirical studies of novice's problems with Prolog show that rank novices have real difficulties in learning the language [van Someren 85; van Someren et al 87; Taylor & du Boulay 87]. Taylor and du Boulay argue that logic and Prolog programming are both subtle and difficult skills and the latter will require both an understanding of Prolog's procedural meaning and Prolog programming and design techniques. Someren's studies of novices' Prolog programs suggest the variability of programs that novices actually wrote and the range of bugs that they made in their programs. Indeed, students in an introductory Prolog course may come from different backgrounds and have a wide disparity in their aptitude and command of programming. Some wrote good programs, while some have only the faintest idea on how to write their first Prolog programs. We argue therefore for the usefulness of a bug diagnosis system that can detect the bugs in novices' Prolog programs, correct them and comment on the code.

The widely used debugging tool for Prolog programmers for all skill levels between novices and experts is the Prolog program tracer. When invoked by the user, the tracer provides an execution trace of a Prolog program. It is not satisfactory for novices as it relies too much on them to debug their own programs. What we want to develop is a teaching tool that has knowledge about the task the novice is attempting and lessen the burden of debugging on the student.

A class of debugging approaches that enhance a Prolog tracer to localise the source of a program error and correct them has been developed for debugging general Prolog programs [Shapiro 83; Lloyd 86; Pereira 86]. In the next section, we will show why these debugging approaches are insufficient for the task of debugging Prolog programs for tutoring purposes.

1.2 Why Shapiro's Bug Diagnosis System is Inadequate

An approach to the error diagnosis of *pure* Prolog programs, which use Prolog's sequential proof procedure, has been first developed by Shapiro [Shapiro 83] to localise the bugs that cause an error of *wrong solution*, *missing solution* or *nontermination* in the program. The user first specifies a goal call which is incorrect and whether the bug is one of wrong solution, missing solution or nontermination. During the interactive debugging session with Shapiro's

bug diagnosis system, the user acts as an *oracle* to answer queries about the results of intermediate subgoals' computation. A solution *S* of a goal is a *wrong solution* with respect to the intended interpretation *I*, if *S* is not contained in *I* but can be derived from the Prolog program. In this case, Shapiro's bug diagnosis system will try to localise a wrong clause instance, that is, a wrong clause instance whose subgoals are all correct with respect to *I* but its conclusion is wrong. A solution *S* of a goal is a *missing solution* in the intended interpretation *I* if the Prolog program finitely fails on *S*. In this case, Shapiro's bug diagnosis system will try to localise the lowest uncovered goal (or uncovered atom), that is, a goal that cannot be derived from the program. *Nontermination* arises in the case that the program loops when it tries to compute a certain goal. In this case, Shapiro's bug diagnosis system will try to localise the error to a wrong definition of a recursive procedure or as a wrong solution that leads into a loop.

Consider the sorting/2 problem and this student's solution of it :

Task : SORTING A LIST

Write a Prolog program sorting/2 which takes a list of numbers as input and sorts the numbers in the list in ascending order. A goal call "sorting(X, Y)" where X is the input list should succeed with Y instantiated to the sorted list. Use the predicate name sorting instead of sort which is an Edinburgh Prolog built-in predicate.

```

sorting([X|T], L) :-
    insert(X, T1, L),
    sorting(T, T1).
sorting([], []).

insert(X, [H|T], L) :-
    X < H,
    append([X], [H|T], L).
insert(X, [H|T], L) :-
    X > H,
    append([H], L1, L),
    insert(X, T, L1).
insert(X, [], [X]).

append(L, [], L).
append([H|Y], L1, [H|Z]) :-
    append(Y, L1, Z).
```

Figure 1-1 : A student's incorrect solution for sorting/2

The bugs in the student's program are :

- the insert/3 subgoal in the recursive clause of sorting/2 should come after the recursive subgoal,
- the base clause of sorting/2 should be sorting([], []),
- the first clause of append/3 should be append([], L, L).

Consider how a Shapiro's bug diagnosis system, together with a model of what the task is supposed to do, would analyse this program :

OK, let's run your program on an example : `sorting([2,3,1,4], X) ...`

*** Error: uninstantiated variable in arithmetic expression: `_375024`
*** Error: uninstantiated variable in arithmetic expression: `_375024`
*** Error: uninstantiated variable in arithmetic expression: `_375069`
*** Error: uninstantiated variable in arithmetic expression: `_375069`
*** Error: uninstantiated variable in arithmetic expression: `_374976`
*** Error: uninstantiated variable in arithmetic expression: `_374976`
*** Error: uninstantiated variable in arithmetic expression: `_374979`
*** Error: uninstantiated variable in arithmetic expression: `_374979`

Your program produces no solution.

{ The system runs the goal call on a model program of `sorting/2` to find out the intended behaviour of `sorting/2`. }

Error : missing solution "`sorting([2,3,1,4],[1,2,3,4])`." Diagnosing ...

*** Error: uninstantiated variable in arithmetic expression: `_375303`
*** Error: uninstantiated variable in arithmetic expression: `_375303`

Now, please answer yes or no based on what you intend your program to do, *not* what your program actually does.

Query : `insert(2, X, [1,2,3,4])?yes`

Which X? **[1,3,4]**

*** Error: uninstantiated variable in arithmetic expression: `_375295`
*** Error: uninstantiated variable in arithmetic expression: `_375295`
*** Error: uninstantiated variable in arithmetic expression: `_375295`
*** Error: uninstantiated variable in arithmetic expression: `_375295`
*** Error: uninstantiated variable in arithmetic expression: `_375295`

Error diagnosed : your program cannot satisfy the goal "`insert(2, X, [1,2,3,4])`".

It has no clause to cover(satisfy) it (*`insert(2, X, [1,2,3,4])` fails when it should succeed*).

Figure 1-2 : Error diagnosis using a Shapiro's bug diagnosis system

Shapiro's error correction system would proceed to try to fix the `insert/3` predicate definition which does not address the real errors in the program. It will propose different perturbations of an `insert/3` clause and ask the student whether to accept or reject it. This example illustrates the general limitations of Shapiro's system for debugging programs for tutoring novices : it relies very much on the structure of the program clauses such as subgoal ordering to anchor the search for the source of errors; it relies on an oracle, namely, the novice to answer queries about the correctness of intermediate results, and to have a fair understanding of Prolog to guide and do part of the debugging himself.

Consider how we might improve the analysis in this example. First, we can consider *modes*, that is, information about whether a predicate's arguments are instantiated when that predicate is invoked. The reader can refer to page 12 for more information on the concept of a mode. The insert subgoal invocation in `sorting/2` has an invocation mode of `insert(+, -, -)` which means that the first argument is instantiated and the second and third arguments are variables when this goal is invoked. Whereas it can be inferred by observing that `'<'/2` and `'>'/2` requires a mode of `(+, +)` and therefore the `insert/3` predicate definition works only for `insert(+, +, ?)`. This would immediately pre-empt an automated debugger to try to solve or execute `insert(2, X, Y)`, or more generally, an insert call of mode `insert(+, -, -)`. Second, in the general case, we can automate the oracle, needed by Shapiro's system to answer its queries about the results of intermediate computations, by using reference predicates which are to be heuristically matched to student predicates. Thus the query about `insert(2, X, [1,2,3,4])`, if there is a reason why it should be pursued, can be answered automatically from a matched reference predicate.

1.3 Why We Need Multiple Sources of Program Analyses

Mode analysis – reasoning with modes – is needed in the Prolog program analysis because a student predicate definition is only correct in solving the task with respect to the intended modes of the predicate. Consider naive reverse :

```
nrev([], []).
nrev([H|T], Answer) :-
    nrev(T, ReversedT),
    append(ReversedT, [H], Answer).
```

The mode `nrev(-, +)` does succeed once, but then loops infinitely on backtracking, because it leads to a call with mode `nrev(-, -)` which leads to a call of `append/3` with mode `append(-, +, -)` which exhibits this behaviour. In Section 8.4, we will explain how `append(-, +, -)` can be inferred to be nondeterminate – returning infinite solutions on successive backtracking. Swapping the goals in the body of the second clause makes `nrev(-, +)` safe and `nrev(+, -)`, the usual call, dangerous.

Type checking can detect the existence of miscoding by detecting Prolog terms of incorrect types in certain predicate positions. A Prolog program is often written under the assumption that all its invocations are of a certain form and do not work properly when called in another way. Consider the one-line program

```
append(X-Y, Y-Z, X-Z).
```

which appends only difference lists and does not work if, for example, the invocation has an instantiated argument which is not a difference list.

Prolog programs are very prone to *mistyping* or *misspelling* errors as such errors do not result in compile-time errors but in programs with unintended meanings. As an example for greater analytic power from interlocking more than one source of analysis, type checking can help to confirm or reject misspelling hypotheses, and vice-versa. *Dataflow analysis* – reasoning with the flow of data in a program – can detect general dataflow anomalies like non-terminating predicates and infer the recursion argument from a predicate definition. All these types of analyses require different forms of expertise which have to be combined to provide possible consistent hypotheses about bugs and corrections to them.

1.4 Why We Need Knowledge of Algorithms

For tutoring to be effective, it is not enough to point out symptoms or program misbehaviour in the student's code such as those detected from the different sources of program analysis. Pointing out an anomaly like a nonterminating loop or a misspelling mistake may lead the student to correct the surface manifestation of the looping behaviour without addressing the main intention of writing a program to solve a given task. It might be that a loop is unnecessary, so the comment about a non-terminating computation is not as apposite as a comment about the inaptness of using a loop. To perform such analyses, task-specific information about the problem has to be used.

If the task is to reverse a list, there are, as in Pascal and in LISP, general plans to follow. One general plan is to detach one element, reverse the remainder recursively and splice the detached element back into the right place. In the *naive* algorithm, it is the first element that is singled out, but it is equally legitimate to take the approach suggested by :

```
PROGRAM 1 :
reverse([], []).
reverse(List, [LastElement|ReversedAllButLast]) :-
    getlast(List, AllButLast, LastElement),
    reverse(AllButLast, ReversedAllButLast).
```

or equivalents, such as the unfolded form :

```
PROGRAM 2 :
reverse([], []).
reverse(List, Answer) :-
    getlast(List, AllButLast, LastElement),
    reverse(AllButLast, ReversedAllButLast),
    Answer = [LastElement|ReversedAllButLast].
```


Another general plan is to use the 'railway shunting' method, using an auxiliary argument as an accumulator. Thus an *algorithm-based* approach is required to be able to recognise the algorithm intended in the program. This requires representing implementations of standard algorithms for doing a task in an automated debugger so that they can be used as the reference to be matched against. We can also use *program transformation* rules to simplify the student program to some canonical form to facilitate matching, for example, folding Program 2 into Program 1. This will provide some parsimony in the representations of task algorithm implementations.

Thus a Prolog program analyser could sensibly use some knowledge of the task and try to see if the student's solution fits in any way. The paradigmatic way to explain discrepancies is to use a bug catalogue. Thus, if there is a mismatch that can be accounted for by any of the canned bugs collected from long observations of students programming the same task, then APROPOS2 will tell the student about it. What and how a Prolog program analyser says to the student is a matter of student modelling, an important issue we are not addressing in depth in this thesis. We propose a debugging approach that provides a PROUST-like account [Johnson 85] of the bugs primarily in terms of their manifestations rather than the underlying conceptions, and student modelling as a step beyond this.

We have not considered Prolog syntax analysis as our interest in this thesis is in the diagnosis of non-syntactic bugs in novices' programs with no syntax errors. Programs with syntax errors will not be successfully consulted or read in by the Prolog interpreter. The use of a syntax or structure editor for Prolog should also preempt syntax errors during program construction.

1.5 A Combination of Debugging Approaches for APROPOS2

Program debugging methodologies are generally of two kinds – *static analysis* that examines the code, and *dynamic analysis* that examines the running of the code on specific examples. For example, Shapiro's bug diagnosis system uses a dynamic analysis approach. In Section 2.5.3, we discuss the advantages and disadvantages of these two approaches. To harness the advantages of both static and dynamic analysis for debugging Prolog programs, we will use a combination of both.

We argue that a combination of debugging approaches coupled with several sources of expertise is important for debugging Prolog programs for tutoring purposes. This contrasts with other recent systems which tend to use just one main debugging methodology.

In this thesis, we will suggest some ways to combine different sources of analysis, and thereby demonstrate their potential for analytic program analysis. We describe APROPOS2 which we have built to experiment with some of these ideas. The debugging approach we use in APROPOS2 is as follows : the student Prolog program is matched against a library of task algorithms for the task. Information about modes, types, predicate names, recursion types, programming techniques used, number of clauses for each predicate definition, clause orderings, and closeness of clause matching are used to select the best algorithm fits and the best implementations of an algorithm. A heuristic best-first search is applied. The result of the search is a set of best algorithm implementation matches for each student predicate definition. Next, a program critic is invoked to try to explain any discrepancy between the student code and the reference code that implements the algorithm. Discrepancies will then be explained by code matching, failing which dynamic analysis is run on parts of the student code using Shapiro's diagnosis routines [Shapiro 83] with matched reference predicates serving to answer oracle queries about the results of the desired execution runs. APROPOS2 relies on this approach to automatically detect and correct nonsyntactic errors in student Prolog programs written to do simple but nontrivial list and number manipulation tasks. Figure 1-3 shows an overview of the debugging process in APROPOS2.

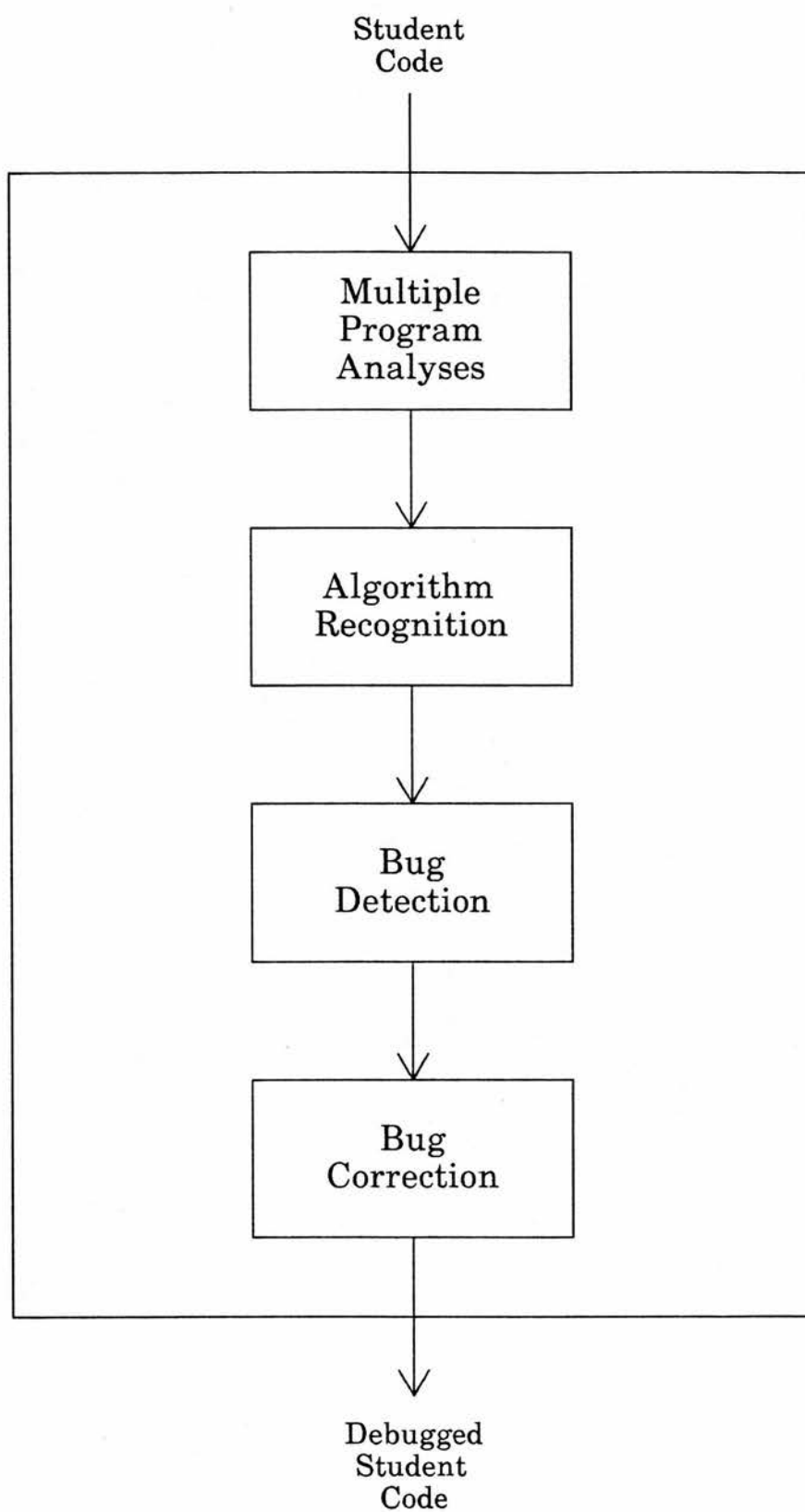


Figure 1-3 : An overview of debugging in APROPOS2

1.6 An Example of APROPOS2's Output

To give an idea of what sort of analysis APROPOS2 provides, consider the buggy sorting/2 program in Figure 1-1. Figure 1-4 shows the result of APROPOS2's analysis. We will discuss the processing behind APROPOS2's analysis in subsequent chapters.

Your program clauses are :

```
sorting([X|T], L) :-
    insert(X, T1, L),
    sorting(T, T1).
sorting([X], [X]).

insert(X, [H|T], L) :-
    X = < H,
    append([X], [H|T], L).
insert(X, [H|T], L) :-
    X > H,
    append([H], L1, L),
    insert(X, T, L1).
insert(X, [], [X]).

append(L, [], L).
append([H|Y], L1, [H|Z]) :-
    appendY, L1, Z).
```

Your program seems to be incorrect.
Now, this is APROPOS2's critique on your program :

Your clause :

```
insert(X, [H|T], L) :-
    X = < H,
    append([X], [H|T], L).
```

can be rewritten as :

```
insert(X, [H|T], [X, H|T]) :-
    X = < H.
```

Your clause :

```
insert(X, [H|T], L) :-
    X > H,
    append([H], L1, L),
    insert(X, T, L1).
```

can be rewritten as :

```
insert(X, [H|T], [H|L1]) :-
    X > H,
    insert(X, T, L1).
```

Do you want to the analysis to proceed ?yes

Clause 1 of insert/3 seems fine.

This clause handles the case when the element to be inserted is smaller than or equal to the first element of the list to be inserted into. It therefore puts the element to the front of the list.

Clause 2 of insert/3 seems fine.

This clause handles the case when the element to be inserted is larger than the first element of the list to be inserted into.

Clause 3 of insert/3 seems fine.

This base case says that the result of inserting an element X into the empty list is the list [X].

Clause 1 of sorting/2 does not seem quite right.

This recursive case sorts the tail of the list and inserts the first element back into the sorted tail. But you should place the recursive subgoal before the insert/3 subgoal.

Clause 2 of sorting/2 does not seem quite right.

Replace `sorting([X], [X])` with `sorting([], [])`.

This base case says that the result of sorting the empty list is the empty list.

Your predicate definition for append/3 is not analysed as it is not used for the main task.

Figure 1-4 : APROPOS2's analysis of a sorting/2 program

1.7 A Formative Evaluation of APROPOS2

At the beginning of this work, we set ourselves two basic criteria for measuring the feasibility of APROPOS2. First, we want to know if APROPOS2 can perform at all, judging from a small number of problems. Second, we want to know whether APROPOS2 handles the range of variability of novice students' programs. We wish to emphasize this aspect of our work as most of the instructional programs for computer programming have not undergone any evaluation and have not got out of the laboratory for any wider use.

We tested APROPOS on four programming tasks. On these problems, APROPOS2 attained an accuracy of 85% for one task, and 95% for the other tasks in correct detection of the algorithm used, and an almost perfect accuracy in detecting bugs, provided it understands the algorithm used. From our empirical experiments, we judge APROPOS2 to be feasible in meeting these two criteria and APROPOS2 to be useful for debugging actual students' Prolog programs.

The intent of our evaluation is not meant to be summative. Rather, the intent is to do a formative survey – to gather data on how APROPOS2 works on real students' programs which can be used to further improve APROPOS2.

1.8 The Contributions of this Thesis

The main contributions of this thesis are :

- (1) A demonstration of a heuristic best-first search of the program decomposition space to parse a Prolog program into a hierarchical structure of predicate definitions, clauses, subgoals, arguments and terms. This thesis shows the use of heuristic code-matching to suggest reference predicates that would match student predicates. This information is used in a program testing approach to find out if running the student predicates on test data would produce the same results as running the reference predicates does. It can also be used to answer oracle queries when debugging using Shapiro's bug diagnosis routines. It shows how static analysis of Prolog programs can be combined with dynamic analysis in order to harness the advantages of both approaches.
- (2) A demonstration of the use of multiple sources of expertise to analyse Prolog programs for the purpose of understanding student programs, detecting bugs and suggesting fixes to buggy student programs. This thesis shows the use of type, mode and dataflow analyses, and misspelling checking in detecting general program anomalies. It also shows the use of type, mode and dataflow analyses, misspelling checking, program transformations and detection of programming techniques in guiding the code-matching process.

We illustrate the merits of an algorithm-based approach supplemented by multiple sources or expertise for program debugging by showing that APROPOS2 is a feasible, realistic and useful tool for program debugging. We highlight the difficulties inherent in debugging Prolog programs, discuss the limitations of our algorithm-based approach and address some of these limitations by proposing a more abstract representation of Prolog programs that permits more flexible code analysis. While we have observed that some students who have used APROPOS2 interactively find the bug reports to be helpful, we make no claims about the pedagogical usefulness of APROPOS2. Indeed, it is intended as the expertise module of PITS and provides the necessary information to help formulate the explanations that a complete PITS can provide as tutorial advice. The research literature reports few evaluation studies of the efficacy and the cost-effectiveness of ITSs for teaching programming, even for recent systems like PROUST and GREATERP [Anderson et al 1987].

1.9 Preliminaries for Prolog

In this section, we briefly describe the terminology for Prolog programming we will use in this thesis. A Prolog program consists of a set of *predicate definitions*. A *predicate definition*

consists of a sequence of clauses. Each clause is a sequence of literals, which are either atomic goals or negations of atomic goals. Prolog clauses are constrained to be definite Horn, that is, they have exactly one positive literal. The positive literal is called the *head* of the clause, and the remaining literals, if any, constitute the *body* of the clause; a clause with only negative literals is referred to as a *goal*. We use the syntax of DEC-10 Prolog and write clauses in the form :

p :- q1, q2, ..., qn.

which can be read as "p if q1 and q2 and ... and qn" and interpreted procedurally as "to satisfy goal p, satisfy goals q1, q2, ..., qn". We use the notation *predicate/n* to denote the predicate definition, the clauses' head of which is *predicate* with *n* arguments.

Our use of the *type* of a predicate is that it describes all terms for which the predicate may succeed. If any term is not described by the type of a predicate, the predicate cannot succeed. We call this the *supportive* type of the predicate. Our types in Prolog will be symbolic descriptions of terms. For instance, the system predicates '=' and '>' has supportive type (*number, number*). Consider the *append/3* program :

```
append([], L, L).
append([H|X], Y, [H|Z]) :-
    append(X, Y, Z).
```

The type of the *append* predicate will then be, in BNF notation :

type *append*(*AnyList*, *AnyList*, *AnyList*)

where *AnyList* = > [] ; [*any*|*AnyList*]

and *any* is a type variable which can be instantiated to any acceptable type. Given a Prolog program and the desired types of the top level predicate(s), the types of how the user-defined predicates are to be invoked in the program can be inferred. Given a Prolog program and the supportive types of system predicates, the supportive types of the other predicates can be inferred. These processes require *type inference*. *Type checking* is the process of checking if any particular invocation of a predicate has types which can be handled by the supportive type of the relevant predicate definition.

The *mode* of a predicate in a Prolog program indicates how its arguments will be instantiated when that predicate is invoked. In this thesis, we adopt the following description of modes :

- o Nothing known about the mode
- The set of variables
- + The set of nonvariable terms
- ? The set of all terms

We distinguish this usage of mode as *calling* or *invocation* modes to distinguish them from *supportive* modes. The supportive mode of a predicate indicates how its arguments must be instantiated when that predicate is invoked – it describes the instantiation states of arguments for which the predicate may succeed. If any invocation is not described by the supportive mode of a predicate, the predicate cannot succeed. For instance, the system predicates ' $=$ ' and ' $>$ ' have supportive mode $(+, +)$.

Given a Prolog program, the desired modes of the top level predicate(s) and the supportive modes of the system predicate(s), the invocation modes of the other predicates can be inferred. Consider the `reverse/2` program :

```
reverse([], []).
reverse([H|T], Res) :-
    reverse(T, Sofar),
    append(Sofar, [H], Res).
```

From information that `reverse` is to be used for mode $\text{reverse}(+, -)$, we infer that the `append/3` predicate needs to work for mode $\text{append}(+, +, -)$. This is called *mode inference*. *Mode checking* is the process of checking if any particular invocation of a predicate has modes which can be handled by the supportive mode of the relevant predicate definition.

Dataflow analysis is a method for analysing internal program behaviour. It views the execution of a computer program as a sequence of events as a flow of data from input to output in which input values contribute to intermediate results, and so forth until the final results, which presumably are output, are obtained [Fosdick & Osterweil 76]. It is the ordered use of data implicit in this process which is the main object of study in dataflow analysis.

1.10 Structure of this Thesis

In this thesis, we describe the debugging approach as incorporated in our current version of APROPOS, APROPOS2. In Chapter 6, we describe an earlier version of APROPOS, APROPOS1 which combines static with dynamic analysis, and some initial work in exploring an analysis-by-synthesis approach using a program called APROPOS-AS. We discuss these earlier work to show how features in APROPOS2's code analysis approach such as

- a general code-matching process which is independent of the task knowledge, and
- a best-first search for the best matches of representations of student programs to representations of reference programs

address limitations of earlier versions which did not incorporate them. We argue that these features are important for APROPOS2's performance in analysing novices' programs.

The structure of the rest of this thesis is as follows :

- The first part of Chapter 2 discusses the general problem of program debugging and the special issues that debugging Prolog programs pose. The second part of the chapter discusses the representations of programming knowledge and task knowledge which have been proposed or are used in existing program construction and debugging tools. The third part is a review of alternative debugging methodologies.
- Chapter 3 provides an overview of the debugging process of APROPOS2. It discusses the two parts of APROPOS2's analysis – performing multiple sources of program analyses and using task-specific knowledge to analyse a Prolog program. We provide an example of APROPOS2's analysis of a student program for reverse/2.
- Chapter 4 describes the frame-like representations that APROPOS2 uses to represent computational features of algorithms and the programs that implement them.
- Chapter 5 explains how the heuristic code-matching process makes use of these representations to identify algorithms in student programs and describes how APROPOS2 automatically corrects bugs it has detected. It describes how we have combined static with dynamic analysis and how we have made use of information derived from the different sources of program analysis.
- Chapter 6 discusses the development history of APROPOS, and describes the earlier versions of APROPOS and their empirical evaluation.
- Chapter 7 describes the empirical evaluation of the current version of APROPOS2.
- Chapter 8 proposes future work on the development of APROPOS2.
- Chapter 9 discusses the role of APROPOS2 in a PITS and proposes future work on the development of a PITS.
- Chapter 10 provides concluding remarks for this thesis.

Four appendices have been included for the reader's reference :

- Appendix I provides brief descriptions of algorithms for the five programming tasks APROPOS2 is tested on.
- Appendix II shows a transcript of a student's session with APROPOS1.
- Appendix III shows a transcript of a student's session with APROPOS-AS.
- Appendix IV shows APROPOS2's analyses of four students' reverse/2 programs.
- Appendix V shows the form used for the independent evaluation of APROPOS2's analyses of reverse/2 programs.

Chapter 2. The Problem of Debugging

In this chapter, we will motivate APROPOS2's approach of combining static and dynamic analyses for the debugging of Prolog programs for tutoring purposes. We highlight the general problem of program debugging and the special issues that debugging Prolog programs pose in contrast to other languages like LISP and Pascal. We discuss the representation of programming knowledge and task knowledge which have been proposed or are used in existing program construction and debugging tools. We also present a review of alternative debugging methodologies.

2.1 The General Problem of Program Debugging

The problem of program debugging is a difficult one because of significant variability in the design and implementation of programs. If the task is to sort a list, several algorithms can be used : permutation-sort, bubble-sort, quick-sort, etc. Each algorithm can be decomposed into procedures in many ways. For each procedure, there can be an infinite number of correct implementations varying in identifiers, programming constructs and data structures used, dataflow, control flow, etc. For each implementation, there can be infinite number of bugs, such as mistyped identifier names, incorrect variable update or instantiations, missing code, unintended side-effects, etc.

The difficult problems in this view of program debugging are :

1. Unanticipated algorithms and implementations of algorithms pose a difficult problem in diagnosing bugs in nontrivial domains.
2. The program may have used an incorrect algorithm. To debug this, we need knowledge of incorrect algorithms or why such an algorithm is incorrect with respect to solving the program task.
3. An incorrect implementation may suggest not one, but one of several algorithms which may be intended but implemented incorrectly. This is not unlike the parsing problem in natural language.

As the general problem of proving program correctness is undecidable, there is no perfect debugging algorithm, no matter what approach we use. The problem is even more difficult for bug detection. This limitation should not prevent us from asking important performance questions. Even human performance fails when the human cannot comprehend the solution. The kind of evaluation criteria we should therefore adopt is : "What percentage of bugs does

this debugger find, and how often does it generate false alarms? How much variability can the debugger tolerate in student solutions without serious performance degradation?" [Murray 86].

We are concerned with debugging for tutoring purposes. This means that besides seeking program correctness in solving a task and locating bugs in programs, an automatic debugger should have enough information to present the bugs to the student modeller of a tutoring system in several possible ways. The tutorial component can then decide how best to describe the bugs to the student. The debugger also needs to suggest possible corrections to the bugs. For this reason, heuristic code-matching has been used as the main debugging approach in ITS for teaching programming, for example, PROUST [Johnson 85], Talus [Murray 86]. An alternative is to use theorem-proving approaches which are more rigorous but fail in actual performance on the real tasks. Even theorem-proving approaches do not provide complete assurance of correctness because the theorem-prover program or its specification is itself susceptible to bugs. To provide an illustration, Goodenough and Gerhart find 7 bugs in a text formatting program informally proved correct by Naur [Shapiro 83]. Our work here adopts a primarily heuristic code-matching approach in analysing Prolog programs supplemented by multiple sources of other program analysis to enable more flexible matching and dynamic analysis to prove program equivalences.

2.2 The Problem of Debugging Prolog programs

The main features of debugging Prolog programs which distinguish it from other programming languages are :

- (1) Prolog is a logic programming language with clear declarative semantics for pure Prolog. This would make verification of programs simpler and easier. But, as Kanamori and Seki noted, there have been few studies that have investigated verification effectively [Clark & Tarnlund 77; Hansson and Tarnlund 79; Tarnlund 81; Kanamori & Seki 86]. Other work has been done in developing algorithms and formal frameworks for the declarative error diagnosis of Prolog programs [Shapiro 83; Lloyd 86; Edman 83; Ferrand 85; Pereira 86; Deshowitz & Lee 87]. These algorithms do not handle the metalogical and the extralogical predicates in Prolog well; for example, Lloyd has provided a sound and complete procedure for debugging only pure Prolog programs.
- (2) A Prolog program can have both a *declarative* and a *procedural* reading. A program debugger can address the declarative interpretation of a Prolog program separately from its procedural meaning. The distinction between the declarative and procedural meaning is not as clear for other languages like LISP and Pascal. While the debugging algorithms mentioned in (1) above can diagnose declarative errors with the programmer only needing to know the intended interpretation of the buggy program to use the algorithms, knowing effectively how to fix the bugs would require knowledge of the procedural

semantics of the language. The effective use of such debugging frameworks would therefore require a fair amount of knowledge about Prolog and as such they are more suitable as general debugging tools than as used for tutoring purposes.

- (3) A Prolog program is very prone to errors that do not result in compile-time errors but in programs with unintended meanings. From the logic programming community, it has been said :

"The procedural semantics of a syntactically correct [Prolog] program is *totally defined*. It is impossible for an error condition to arise or for an undefined operation to be performed. ... A totally defined semantics ensures that programming errors do not result in bizarre program behaviour or incomprehensible error messages." [Warren, Pereira & Pereira 77], *italics ours*.

"The totally defined semantics of a Prolog predicate leads to a different interpretation with respect to errors. As *normal* values cannot be distinguished from *error* values, errors are extremely difficult to detect and understand. This is particularly the case when a simple mistyping or misunderstanding concerning the nature of the predicate is involved." [Mishra 84]

- (4) A Prolog program may or may not be *multi-directional*. A program debugger needs to explore what input and output modes the program may be correctly used for. Consider this intersect/3 program which was originally in the DEC-10 Prolog library :

```
intersect([], _, []).
intersect([H|T], L, [H|T1]) :-
    member(H, L),
    !,
    intersect(T, L, T1).
intersect([_|T], L, T1) :-
    intersect(T, L, T1).
```

which was intended to work for mode intersect(+, +, ?) but does not work properly when the invocation is of mode intersect(+, +, +). For example, a call of

```
?- intersect([a,b], [a,b], []).
```

will succeed.

- (5) Prolog programs can be *nondeterminate*. A program debugger may have to take into account all the solutions which can be returned from a Prolog program when called in a certain mode. Consider the program nrev :

```
nrev([H|T], Res) :-
    append(Sofar, [H], Res),
    nrev(T, Sofar).
nrev([], []).
append([], L, L).
append([H|L1], L2, [H|L3]) :-
    append(L1, L2, L3).
```

which loops after producing one answer when invoked with mode nrev(+, -). If the order of the two append/3 clauses are reversed, then nrev(+, -) will loop before even producing an answer.

- (6) The computation model of Prolog is based on goal invocation, and goal success and failure. Thus errors in Prolog programs occur when, for example, they finitely fail on goals that should succeed, or succeed on goals that should fail.
- (7) Prolog programs run on a model of computation that involves unification and depth-first search. A program debugger needs implicit knowledge about these aspects of Prolog computation to be able to analyse Prolog programs and explicit knowledge to be able to explain them.
- (8) A Prolog program may have used a number of *programming techniques* specific to Prolog. An example of such a technique is the use of an argument position to accumulate the result recursively and the use of another argument position to instantiate the final result as in reverse/3 :

```
reverse([], L, L).
reverse([A|X], Y, Z) :-
    reverse(X, [A|Y], Z).
```

A program debugger can attempt to recognise the use or incorrect use of such techniques to understand the intent behind the student code. The automatic detection of the use of techniques in a student's program is a difficult task because the student may have used a technique incorrectly or used an inappropriate technique. An automated debugger needs some expectation of what the student is intending to do which can be partially inferred at least from reasoning about modes.

- (9) In Prolog there are few system predicates which might serve as markers for heuristic template-matching.

We were concerned with the above issues when we started our work in developing a program analysis approach for novice Prolog programs. (1), (2), (4), (5), (6), (7) and (9) suggest that we need dynamic analysis of Prolog programs. (2), (3), (4) and (8) suggest that we need static analysis of Prolog programs. (1) and (9) suggest a formal program verification approach but we will discuss in Section 2.5.2.3 why this is currently not a feasible approach. (3) suggests the usefulness of misspelling analysis while (4), (5) and (8) require mode analysis. Consideration of these issues influenced us in choosing a primarily heuristic code-matching approach to analyse Prolog programs for a small class of problems for tutoring purposes, and in using multiple sources of analyses such as reasoning with modes and mistyping to help the code-matching process.

2.3 Representations of Programming Knowledge

Intelligent Tutoring Systems (ITSs) for teaching programming languages need to use programming knowledge to reason with students' programs, detect the bugs in them, suggest corrections, and explain the bugs and their likely causes to students. One key component of these ITSs is the representation of knowledge about programming and programming

solutions. We review some of the different representations of programming knowledge used in current ITSs, program debugging and program construction tools, and discuss how they might be applicable to Prolog. We describe the concept of Prolog programming techniques and discuss why a technique-based approach is insufficient. We also motivate an algorithm-based approach for debugging Prolog programs in PITS.

Studying representations of programming knowledge also allows us to explore epistemological issues concerning the nature of knowledge that is used for programming and how that knowledge can be learned and thus how it can be effectively taught, although this is not our main concern in this thesis. The long-term pedagogical objective of research into ITSs for programming is to improve the quality of computer programming education. For instance, Soloway and his co-researchers have provided empirical evidence that Pascal programmers use a lot of *plans* in programming activities [Soloway & Ehrlich 84], and have argued for more explicit teaching of programming plans as a basis for a programming curriculum instead of the traditional emphasis on the syntax and the semantics of the language.

Our main interest here is the kinds of knowledge programmers use when they write programs, as contrasted with the kinds of knowledge programmers used for debugging programs. Work done in this latter area includes the study of students' debugging of LOGO programs [Carver & Klahr's, in press], and the study of students' debugging of LISP programs [Kesler & Anderson 86]. The question of whether common knowledge structures are used for the writing, comprehension and debugging of programs is an interesting issue that requires further study but is beyond the scope of this thesis.

2.3.1 Prolog Programming Techniques

In this section, we provide some examples of Prolog programming techniques, describe some problems with a technique-based analysis and discuss why such an approach is insufficient for analysing Prolog programs. By programming techniques in Prolog, we mean the use of common implementation methods specific to the Prolog language which can be brought to bear on a wide range of problems. Generic techniques concerned with simple list processing include [Ross 87] :

- smashing a structure recursively to find a desired part, or until some part satisfies a required test (for example, member/2)
- smashing a structure recursively while building another from selected parts of the first or from information about the first
- using backtracking generatively for generate-and-test
- using auxiliary arguments for communicating data or 'results'
- incrementally instantiating the parts of a desired structure

There are many dimensions to Prolog programming techniques. Consider these techniques which are concerned with constructing data structures :

- Building the result using an accumulator during invocation of recursive calls.

Example : *flatten/2* to flatten a list
 % third argument of *flatten/3* accumulates the result
 % as recursion unfolds
 flatten(List, Res) :-
 flatten(List, [], Res).
 flatten([], Res, Res) :-
 !.
 flatten([Head|Tail], Sofar, Res) :-
 !,
 flatten(Head, Sofar, MoreOfResult),
 flatten(Tail, MoreOfResult, Res).
 flatten(NonListStructure, Sofar, [NonListStructure|Sofar]).

- Building the result using an accumulator after invocation of recursive calls.

Example : *flatten/2* to flatten a list
 % second argument of *flatten/3* accumulates the result
 % as recursion folds
 flatten(List, Res) :-
 flatten(List, Res, []).
 flatten([], Res, Res) :-
 !.
 flatten([Head|Tail], Res, Sofar) :-
 !,
 flatten(Head, Res, TailResult),
 flatten(Tail, TailResult, Sofar).
 flatten(NonListStructure, [NonListStructure|Sofar], Sofar).

Consider these techniques which are concerned with the use of the *cut* for different purposes to affect the procedural behaviour of programs :

- Use of the cut to confirm the choice of a rule.

Example : *head :-*
 tests,
 !,
 actions.

- Use of the cut in enumerating cases.

Example : *head :-*
 test-for-case-1,
 !,
 action1.
 head :-
 test-for-case-2,
 !,
 action2.
 ...
 head :-
 !,
 actionk.

- Use of the cut to avoid deep recursion and to utilise tail-recursion. The removal of the cut from the program here does not affect its procedural meaning.

Example : to recognise a sequence of letters and digits.

```
word([Ch|Chs]) :-
    ( letter(Ch) ; digit(Ch) ),
    !,
    word(Chs).
word([]).
```

Consider these techniques which are concerned with *iteration* or *recursion* :

- Iterate through a whole structure, like a FOR loop in Pascal.

Example : append([], Y, Y).
 append([A|X], Y, [A|Z]) :-
 append(X, Y, Z).

- Iterate until a terminating test succeeds, like a WHILE or REPEAT loop in Pascal.

Example : member(X, [X|_]).
 member(X, [_|Y]) :-
 member(X, Y).

One observation here is that if the argument that controls or runs the iteration unifies in the different clauses and has a '-' mode declaration, then the terminating case should come before the recursive one. In this member/2 program, it is the second argument that controls the iteration. Thus a goal call of mode member(+, -) will succeed first by matching with the first clause. On subsequent backtracking, it matches the second clause which will produce infinitely many solutions. However, if the recursive clause is placed before the terminating case, a goal call of mode member(+, -) will first match with the recursive clause leading to an infinite chain of invocations with mode member(+, -). The general points here are that mode information must be brought to bear in the judicious use of techniques, and that the procedural semantics cannot be ignored.

- Iterate through solutions until a terminating test succeeds.

Example : earliest_ancestor(X, Y) :-
 ancestor(X, Y),
 not(ancestor(Y, _)).

- Iterate by use of fail/0 and backtracking (iterate exhaustively).

Example : retractall(X) :-
 retract(X),
 fail.
 retractall(X) :-
 retract((X :- Y)),
 fail.
 retractall(_).

- Iterate by use of repeat/0 (repeat until a terminating test succeeds).

Example : repeat_get(+X) to get the next character in the input stream that matches X.

```
repeat_get(X) :-
    repeat,
    get0(X).
```

The use of programming techniques for teaching, programming, as the basis of a programming editor, in debugging tools, in ITSs and language design is discussed in [Bundy et al 87]. Our interest here is debugging Prolog programs in a PITS. Our viewpoint is that while programming techniques have an important role to play in an ITS for Prolog especially in the tutorial component since program techniques are what we want to teach, a technique-based approach is itself not sufficient for bug diagnosis. The automatic detection of the use of techniques in a student's program is a difficult task because the student may have used a technique incorrectly or used an inappropriate technique. An automated debugger needs some expectation of what the student is intending to do. Consider these programs :

PROGRAM 1 :

```
append([], L, L).
append([A|X], Y, [A|Z]) :-
    append(X, Y, Z).
```

PROGRAM 2 :

```
reverse([], L, L).
reverse([A|X], L, Y) :-
    reverse(X, L, [A|Y]).
```

Both these programs use the first argument as the iterative counter – a plan of using a list to control the recursion. Used in the usual mode sense append(+, +, -) (for example, a call of append([a,b], [c,d], Result) to find the Result of appending [a,b] to [c,d]), Program 1 builds up the output structure in the third argument of the head goal. This technique builds the intended result in the head goal. Used with mode reverse(+, -, +) (for example, a call of reverse([a,b], Result, []) to find the reverse of [a,b] and put it in Result), Program 2 builds the intended structure in the third argument of the recursive goal which functions as an accumulator. This suggests that mode information as well as other intentions of the programmer must be brought to bear for debugging. If techniques are identified without knowledge about intentions, then buggy and correct code cannot always be distinguished. Thus, if the student's code is :

```
foo([], L, L).
foo([A|X], Y, Z) :-
    foo(X, Y, [A|Z]).
```

without reasoning with intentions, an automated debugger cannot distinguish whether the program is correct code for reverse/3 or buggy code for append/3 or whether the programmer has intended to build the result structure in the head goal but realised it incorrectly, or has correctly used a plan to build the result structure in the recursive subgoal.

A technique-based approach in an automated programming tutor for Prolog would rely on technique-fitting against a set of prestored techniques to criticise the program on the basis of its fit to the technique. Because the syntactic closeness of quite different techniques, it is difficult for an automated debugger to know which technique is intended when it is incorrectly or inappropriately used. Even if it can recognise the intended technique, criticising the incorrect use of the technique may be inappropriate in relation to solving the task. For example, if the student used an incorrect algorithm that employs the technique. In our program analyser APROPOS2, a few programming techniques are recognised in relation to the algorithm they help to implement thus contributing to the measure of its match to the student's algorithm. We are not however de-emphasizing the usefulness of teaching programming techniques to students. We argue that representations of Prolog programming techniques will be more useful in a technique-oriented editor than in an automated debugger in as much as that plans and cliches in the Programmer's Apprentice discussed in Section 2.3.3 are more suitable as tools for program construction than for program debugging.

2.3.2 Programming Plans

The concept of programming techniques is different from the concept of plans as used in the work of Soloway and his co-researchers at Yale. They consider programming goals and plans as the two key components in representing problems and programming solutions. Programming plans describe stereotypic action sequences in programs and are intended to be language independent. The notion of plans in Yale has been derived from empirical observations that programmers use programming plans in understanding and writing programs. The claim is that programmers write programs by determining what goals and constraints must be satisfied, and then selecting plans which satisfy these goals and constraints. Empirical studies which provide evidence for this claim are described in [Soloway, Ehrlich, Bonar & Greenspan 82], [Soloway, Bonar & Ehrlich 83], [Ehrlich & Soloway 83] and [Soloway & Ehrlich 84].

PROUST is a computer program developed at Yale that analyses Pascal programs written by novice programmers, detects the bugs, and explains them and their causes to the programmers. It embodies the view that programmers use plans to map their intended goals into code. PROUST's task is thus an attempt to invert this mapping, mapping the code back to the programmer's goals. The mapping process involves selecting plans which maps goals to code. The representation of a plan in PROUST is a frame with a set of slots. The main part of the plan is the template which is fairly close to the Pascal code they are matched against. This is PROUST's representation of the plan for SENTINEL PROCESS READ WHILE [Johnson 85]:

```

Constants :    ?Stop
Variables :    ?New
Template :
  InitInput :   subgoal Input(?New)
  MainLoop :    while ?New <> ?Stop do
                  begin
  Process :      ?*
  Next :         subgoal Input(?New)
                  end

```

Parts of the template, such as `while ?New <> ?Stop do ...`, are Pascal expressions. Other parts, such as *subgoal Input(?New)*, are goals which are to be implemented using other plans. The part `?*` matches an arbitrary sequence of statements. The **Constants** and **Variables** slots are plan annotations indicating the types associated with the pattern variables. `?Stop` and `?New` are pattern variables which match to the actual Pascal constant and variable names used in the program.

Consider the averaging problem for Pascal novices [Soloway 86]:

"Write a program that will read in integers and output their average. Stop reading when the value 99999 is input."

The four requirements of the task are considered the goals of the problem. The goals are to compute the sum, to divide the sum by a count of the numbers added, to input the numbers and to output the average. The goal of computing the sum can be realised by a "Sentinel Controlled Running Total" plan which is represented as a template-like structure for reading in a stream of integers, summing them and stopping when a sentinel value (99999) is input.

The plan is as follows :

```

initialise a running total
ask user for a value
if input is not the sentinel value
then  add new value into running total
      loop back to input

```

This is the code realisation in Pascal :

```

sum := 0;
read(new);
while new <> 99999 do
  begin
    sum := sum + new;
    read(new)
  end;

```

We now attempt to implement this plan in Prolog. Both these programs can be code realisations of the same plan :

PROGRAM 3 :

```
computesum(Total) :-  
    computesum(0, Total).  
computesum(99999, 0) :-  
    !.  
computesum(New, Total) :-  
    read(New1),  
    computesum(New1, Total1),  
    Total is Total1 + New.
```

PROGRAM 4 :

```
computesum(Total) :-  
    read(New),  
    computesum(New, 0, Total).  
computesum(99999, Total, Total) :-  
    !.  
computesum(New, Sofar, Total) :-  
    NewSofar is Sofar + New,  
    read(New1),  
    computesum(New1, NewSofar, Total).
```

Program 3 uses the technique of *naive recursion*, while Program 4 uses the technique of *tail recursion* in which the second argument is used to accumulate the result and the terminating case instantiates the third argument to the final accumulated result. As the mapping between the plan and Program 3 or Program 4 is not as direct as between the plan and Pascal code, there is a finer distinction to be drawn besides identifying what programming plan(s) may be used in a piece of code. The distinction is that different techniques may be used to implement the same plan. Thus Prolog programming techniques such as these tend to be more generic in the sense that they are applicable across a wide range of plans.

Besides incorporating knowledge about programming plans, PROUST also represents knowledge about bugs. Two kinds of bug knowledge are represented. In PROUST, most bugs are detected when differences are found between expected code and actual code. These bugs are codified as plan-difference rules which are triggered on certain types of mismatches and they suggest how to interpret these mismatches, either as bugs or correct implementation variants. These rules are derived from extensive observations of the kinds of bugs Pascal novices make. Johnson [Johnson 85] provides this example of a plan-difference rule that is triggered if some variable initialisation is missing from the actual code to explain the bug as one of missing initialisation :

```
IF there are no match candidates  
  AND the plan component is Init:  
  AND some Init: components are present in previously-matched plans  
THEN  
  mark the Init: component as being accidentally omitted.
```

The other kind of bug knowledge is buggy plans which incorporate bugs in them. The following is an example of a buggy plan which is supposed to check input for validity. It uses an IF statement rather than a loop, so it does not work correctly if two invalid data are successively read in [Johnson 85] :

Variables : ?Val, ?Pred

Template :

```
Guard :      IF ?Pred THEN
              subgoal Output Diagnostic()
            ELSE
              ?*
```

While we believe that Prolog programmers also use a lot of plans in their programming activities, the notion and representation of plans in Prolog are subjects for further study. A template-like representation of plans as in PROUST is suitable only for imperative sequential languages like Pascal where there are keywords to anchor the analysis. In Prolog where the model of computation is based on unification, a template-matching process has to be more flexible, for example to recognise different but equivalent unfolded forms of a Prolog expression as equivalent.

2.3.3 Cliches

The key representations of programming knowledge used in the Programmer's Apprentice (PA) project at MIT [Rich 81] are similar to those of Soloway's plans. In the PA's work, the plan formalism is used to represent two basic kinds of information : the structure of particular programs expressed essentially as a flow chart where data flow and control flow are represented by explicit arcs, and knowledge about *cliches*. The goal of the PA work is to develop a computer assistant that acts as a junior partner and critic to a programmer who concentrates on the harder parts of the process. The PA is intended to assist in the easy parts of the programming process by keeping track of the critical details which make a program work.

In the proposal of the PA, the programmer specifies an *underlying plan* which describes using a standardised vocabulary the meaning of the different parts of the program. He then implements his plan in LISP. The PA checks that the LISP code is compatible with the programmer's intention, by constructing a *surface plan* describing the flow of data and control, and then by recognising the correspondence with the help of program annotations between the segments of the surface plan and those of the underlying plan. While this can be an useful tool for the programmer, it requires him to provide the system with three versions of the same program – the plan, annotations and the actual code [Wertz 87].

Waters [Waters 85] proposes a different approach to the programmer's apprentice : instead of entering the different versions of a program, the programmer will work within a cliché editor and the translation of clichés into code will be done automatically. The one demonstration system to emerge out of the PA project is the Knowledge-Based Editor in Emacs (KBEmacs) which only focuses on the task of program construction. KBEmacs is a program editor that allows a programmer to specify the algorithm structure of a program rather than its textual or syntactic structure. It is able to construct a program out of algorithmic fragments which it knows of.

A cliché is defined as a standard method for doing a task, a partial solution of some sort [Waters 85]. We see the cliché in the PA as a high-level template that represents the dataflow and control flow of an algorithm where the place-holders are the "roles" of the task. Roles vary from one use of the cliché to the next but have well-defined purposes. Here is an example of a straightforward cliché called average which computes the average of two input numbers taken from [Waters 85] :

```
(DEFINE-CLICHE AVERAGE
  (PRIMARY-ROLES (X Y)
    DESCRIBED-ROLES (X Y)
    COMMENT "computes the average of {the x} and {the y}")
  (// (+ {the input x} {the input y}) 2.0))
```

This is another cliché called list-enumeration which enumerates the elements of a list :

```
(DEFINE-CLICHE LIST-ENUMERATION
  (PRIMARY ROLES (LIST)
    DESCRIBED-ROLES (LIST)
    COMMENTS "enumerate the elements of {the list}")
  (LET* ((LIST {the input list}))
    (LOOP DO
      (IF ({NULL, the empty-test} LIST) (RETURN))
      {{{CAR, the element-accessor} LIST}, the output element}
      (SETQ LIST ({CDR, the step} LIST))))))
```

The concept of the PA's plans and clichés is similar to the concept of programming plans as used by Soloway. One difference is that Soloway's plans are organised round the goals they implement. Cliches provide a vocabulary of relevant intermediate and high-level concepts for the programmer to interact with the PA. It facilitates program construction by allowing the programmer to specify his program in terms of clichés chosen from a library of prestored clichés.

The idea of using clichés for program construction is intended to be language independent although clichés themselves are not language independent. To support a new language, a different library of clichés which are appropriate for writing programs in that language has to be provided. A separate library is required because programming languages differ

semantically as well as syntactically. For example, the list-enumeration cliché above has to be rewritten for Prolog.

The issue of which algorithms can be expressed as a cliché and which cannot be has not been investigated in great detail [Waters 85]. The dataflow and control flow of the cliché is intended to be nonvarying, with all the variability confined to what fills the roles. The number of roles and the way they interact are fixed in a cliché, analogous to the situation in many programming languages in which the number of arguments to a subroutine call is fixed. Unfortunately, there are algorithms which do not have a fixed number of roles. For example, it is easy to define the cliché which computes the average of two numbers or a different cliché which computes the average of three numbers. However, it is not possible to define the cliché which is the generalisation of these two clichés.

The concept of using clichés as a tool for program construction is practical but the use of a cliché library to recognise clichés from constructed programs falls short. As Waters notes :

"From the point of view of the PA project as a whole, the most glaring defect of the analyser module is the fact that it is not able to recognise which clichés could have been used to construct a program.

When a program is constructed by means of knowledge-based editing using clichés, then the plan contains a record of the clichés which were used. Several of the capabilities of KBEmacs depend on the presence of this information. Unfortunately, since analysis is not able to recover this information, these commands cannot be used after a program has been modified textually. " [Waters 85, p. 179]

Thus while it is relatively easy to convert clichés to code, the inverted process is a many-to-one mapping which is difficult to recover. In this aspect, the PROUST system is reasonably successful in recovering the inverted mapping through a sophisticated process of heuristic code-matching, as demonstrated for a few programming problems.

The concept of the cliché as a common theme in Prolog programming has been discussed in [Lynch 86; Bundy 86]. The idea stems from the observation that there are superficial features of what a Prolog program should look like. Thus an example of a cliché that occur very frequently in Prolog programs is that a program that uses lists may have clauses of the form :

functor([], ...).

and

functor([Head|Tail], ...) :-
 functor(Tail, ...).

Bundy distinguishes *prototypes*, *templates* and *schemata* as all being types of clichés but are clichés of increasing generality. He defines a prototype as a particular Prolog program. To match a program against a prototype would require a set of mappings of predicate and variable names. A template is defined as a generalised prototype in which particular predicate and variable names are replaced with second-order variables. A schema is defined

as a generalised template that allows more variability to be specified, for example, a variable number of arguments, literals or clauses. Bundy does not think that a language for describing schemata is possible owing to the many dimensions of variability. Bundy also contrasts Prolog programming techniques as descriptions of parts of programs from cliches which are descriptions of complete procedures. [Lynch 86] initially proposes using a syntactic cliché-matching approach in a tutoring system for Prolog, while [Bundy 86] subsequently proposes a technique-based approach. A syntactic cliché-matching approach would rely on template-matching against a set of prestored templates for the task. The main difficulty is that there may be too many such templates to enumerate. In our program analyser APROPOS2, we use representations of equivalence classes of templates with respect to computational equivalence and had APROPOS2 reason about them using knowledge about equivalent Prolog predicates and dynamic analysis. The representation we have chosen for APROPOS2 falls somewhere between templates and schemata, as we use second-order variables in our templates and allow for reordering of arguments within predicates, reordering of clauses, and missing or extra literals and clauses. Our code-analysis approach allows for the program transformations of folding and unfolding, and for the computational equivalence of syntactically-different code, thus minimising the need to have a prohibitive number of templates.

2.3.4 Production Rules

The representation of programming knowledge as production rules has been used in Anderson's GREATERP tutor for beginning LISP programming [Reiser et al 85]. The tutor is based on the ACT* theory of cognition which makes claims about the organisation and acquisition of complex cognitive skills such as those required for computer programming. ACT* and its successor theory PUPS make an important distinction between *declarative* and *procedural* knowledge [Anderson et al 87]. Declarative knowledge is what gets stored in human memory when someone is told something, as in instruction or reading a text. Procedural knowledge is what can only be acquired from the use of the declarative knowledge through practice, and embodies knowledge in a highly efficient and ready-to-use way. The ACT* theory claims that procedural knowledge is created from the use of declarative knowledge in problem-solving.

In the PUPS architecture, declarative knowledge is represented in schema-like structures. These structures are thought to be what are acquired by the student before they are later compiled into procedural knowledge. [Anderson et al 87] provides an example of how an ideal student might encode this piece of textual information about the CAR function :

The value returned by CAR is the first element of the list given as its argument.
 (CAR (FAST COMPUTERS ARE NICE))
 FAST

There are two forms of knowledge : abstract and specific. The abstract encoding of **CAR** has a pointer to **structure** that indicates how **CAR** is to be used. The **example** structure has the same form as **structure**, except that an argument is specified. Two other structures encode that argument and the value returned by the example call.

CAR	ISA	: function
	FUNCTION	: (implements first)
	FORM	: (text CAR)
	PRECONDITION:	(type structure)
structure	ISA	: lisp-code
	FUNCTION	: (calculate (first arg))
	FORM	: (list CAR arg)
example	ISA	: lisp-code
	FUNCTION	: (illustrate CAR) (calculate (first lis))
	FORM	: (list CAR lis)
lis	ISA	: list
	FUNCTION	: (argument-in example) (hold (FAST COMPUTERS ARE NICE))
	FORM	: (list ' FAST COMPUTERS ARE NICE)
fast	ISA	: atom
	FUNCTION	: (value-of example) (first lis)
	FORM	: (text fast)

Procedural knowledge is represented by a set of production rules that define the skills necessary for programming. The aim of the tutor is to create experiences that will cause students to acquire such production rules. [Anderson et al 87] provides two examples of production rules :

PRODUCTION RULE 1 :

IF the goal is to merge the elements of lis1 and lis2 into a list
 THEN use *APPEND* and set as subgoals to code lis1 and lis2

PRODUCTION RULE 2 :

IF the goal is to code a function on a list structure, and
 that function must inspect every atom of the list structure, and
 the list structure is arbitrarily complex
 THEN try *CAR-CDR* recursion and set as subgoals
 1. to figure out the recursive relation for *CAR-CDR* recursion
 2. to figure out the terminating cases when the argument is nil or an atom.

Starting off from some programming goal, appropriate production rules are applied to decompose the goal into subgoals until goals are reached which can be accomplished by direct translation into code. GREATERP attempts to model the student's programming knowledge

in terms of a set of correct and incorrect or buggy production rules he is thought to use when programming. [Reiser et al 85] provides a related buggy rule related to the Production Rule 1 above that applies to code the function *APPEND* is :

```
IF      the goal is to merge the elements of lis1 and lis2 into a list
THEN   use LIST and set as subgoals to code lis1 and lis2
```

with an associated template commentary :

You should combine the first list and the second list. But *LIST* is not the right function. If you *LIST* together (a b c) and (x y z), for example, you will get ((a b c) (x y z)) instead of (a b c x y z). *LIST* just wraps parens around its arguments.

The current version of the tutor has about 500 correct rules and 700 buggy rules [Corbett 87]. The buggy rules are derived from observing the errors students make when they use the tutor, trying to understand the origin of the errors and then coding the inferred buggy productions into the system. Thus the programming knowledge of the student is modelled in terms of a set of executable rules.

To use production rules as a model of the knowledge used for Prolog programming, we need to at least rely on work on the representation of Prolog programming techniques or plans, since techniques and plans are what the action parts of the rules are to implement. For example, we can write an equivalent rule for Prolog similar to Production Rule 2 above for LISP.

2.3.5 Algorithms

Our approach to Prolog code analysis is to view the task as uncovering the mapping from algorithm to its decomposition into Prolog code. The approach synthesizes programs in the space of possible programs by first selecting an algorithm and then decomposing the algorithm into a procedure that invokes subtasks, subtasks into further subprocedures and eventually into code. Program analysis will rely on attempting to understand the intentions of the programmer as to what algorithm he intends and how he intends to implement it.

The approach can be seen in the light of our view of the learning phases a Prolog novice goes through. In the first few weeks of Prolog, a student has to cope with learning about the new syntax and semantics of the new programming language. His main preoccupation would be the implementation of common algorithms into Prolog code. Indeed, most Prolog texts concentrate on the ways of converting some algorithm into known Prolog formalism [Sterling & Shapiro 86; Bratko 86]. After the first few weeks of cognitive disorientation with a new language, the main preoccupation shifts to the task of turning some problem statement into an algorithm. Our intention is that APROPOS2 will be used in the initial learning phase of the Prolog novice to lead students through fairly standard exercises of list and number

manipulation. Our approach is also suitable for debugging bugs at the different levels of abstraction which will be discussed in Section 3.2. We also relate our approach to the use of Prolog programming techniques. For tasks and subtasks are mapped to code by selecting and using appropriate programming techniques. The wrong use of a technique or the use of an inappropriate technique will result in a mapping of subtasks to incorrect procedures or code.

Our view concurs with the empirical observations of van Someren [van Someren 84]. He describes two approaches to programming in Prolog :

- "logic programming" in which a program is written as a theory, formulated in logic, and designing the program involves defining relations and properties;
- "algorithmic programming" in which a program is written as an implementation of some algorithm, like programming in some instruction-oriented language.

From studies of about 50 buggy Prolog programs and a talk-aloud protocol analysis of a student writing a program for maximum/2, van Someren observed that the first method of Prolog programming is not common among the novices he observed. However, we should also take note that student behaviour is a function of the way they are taught.

The "algorithmic programming" process of Someren consists of the following steps :

1. Design an algorithm
2. Translate the algorithm into Prolog
3. Verify the program by executing it (mentally or on a machine) on some examples.

The two approaches may not be really separate. For designing an algorithm to a Prolog programmer may really be designing an algorithm *appropriate to the Prolog execution model* and this is where novices have much difficulty. Indeed, this is the viewpoint we adopt of algorithms. We will discuss this in more detail in Section 4.2.

van Someren's view of programming allows us to break down the programming process for further study. It also supports one classification of Prolog bugs naturally. The task of a bug diagnosis system would be to identify both the algorithm and the implementation of it in Prolog and present both to the student. This view of diagnosis is similar to the analysis-by-synthesis approach proposed by Johnson [Johnson 85]. We discuss classifications of Prolog bugs in the next section.

2.4 Classifications of Prolog Bugs

van Someren provides a classification of bugs by their manifestations [van Someren 84]. Some of the bugs he observed in the study of some 50 buggy students' programs are :

1. Use of predicates as functions in an argument position – predicates are thought to return a non-truth value
2. Erroneous base case
3. Erroneous recursive call
4. Identical variable names as global variables.

An example of a program that manifests bugs 2, 3 and 4 is :

PROGRAM 5:

```
last([Hd|Tl], L) :-  
    last(Tl, Hd).  
last([], L).
```

van Someren's view of programming suggests this classification of the causes of bugs – bugs resulting from buggy algorithms, coding errors from buggy implementation techniques, and verification bugs from misconceptions about the Prolog machine. This is a good classification of bugs for pedagogical purposes as it distinguishes the different causes of Prolog errors. To illustrate, in van Someren's view, the cause of the bugs in Program 5 is one of verification, that the student tried to implement a certain algorithm and believed his program works in a certain way, more specifically, that :

1. The last action of Prolog is a call of last/2 with the empty list as the first argument and the temporary last element as the second argument.
2. After the last action, the value of the second argument is transferred to the second argument of the original query [van Someren 84].

Section 3.2 describes the kinds of bugs APROPOS2 detects. We have not yet begun to seriously study verification or conceptual bugs as we believe more empirical work on the types of misconceptions and relating them to their bug manifestations are matters for further research. In this thesis, we identify bugs in behavioural terms and this is at least necessary in a PITS that is able to comment and criticise students' solutions and answer students' queries. There is practical use for such a debugging system, as attested by a scenario in which a student after some futile attempts may ask for the solution to programming a certain predicate in which case PITS can provide a solution that is closest to the student's incorrect attempt. PROUST is one of the few systems that attempt to trace bugs to their underlying misconceptions but unfortunately does not seem to be sufficiently tested on a wider range of programming problems. Besides, PROUST does no student modelling, and so the same bug

observed in programs written by different students is always explained by the same misconception. Its approach is reasonable as the relating of different bugs to misconceptions has been derived from previous empirical observations of students' programs written for the same tasks that PROUST is built to analyse, but it is limited.

A classification of Prolog bugs was proposed in [Brna et al 87] with the different aim of motivating new and improved programming tools for Prolog environments. It classifies an error in a Prolog program at different levels of description :

- the Symptom Description : the symptoms presented to the user
- the Program Misbehaviour Description : the explanation offered for a symptom
- the Program Code Error Description : the explanation offered in terms of the code itself
- the Misconception Description : the underlying conceptualisation in the programmer's mind that caused the error.

Brna illustrates from this example of a similar program as Program 5 [Brna 86a] :

PROGRAM 6 :

```
last([], Answer).  
last([H|T], Answer) :-  
    last(T, Answer).
```

The symptom description is that the answer is a variable when it should be the last element of the list. The program misbehaviour description is that the call to last/2 with the first argument as a list of one element should have succeeded on the first clause rather than failing and should have also bound the second argument to the required answer. The program code error is that the first clause should be replaced with

```
last([Answer], Answer).
```

Brna describes some possibilities for the misconception that gives rise to this bug which can only be inferred from an analysis which must at least include modelling of the student's performance on a longer term basis.

Brna's bug classification suggests that there may be different levels of description of a program bug, some of which may be more readily understood by the student with his current level of understanding and experience of Prolog. Our view is that knowing the program code errors is necessary to be able to correct the bug(s) and provides help as to how to generate and explain the symptom and program misbehaviour descriptions. If, for example, the error is a mistyping one, then it is useful just to pinpoint the slip without bothering about how the program behaves, as in this example of a program that is intended to print all elements of a list :

The program clauses analysed are :

```
writelist([]).  
writelist([Element|Elements]) :-  
    write(Element), nl,  
    writelist(elements).
```

In

```
writelist(elements)  
the argument "elements" is an atom but what is required is a list.
```

Declarative debugging approaches are concerned with diagnosing these classes of bugs :

- nontermination of a predicate invocation
- partial correctness/soundness of a program : does the program succeed on goals for which it should fail?
- completeness of a program : does the program fail on goals for which it should succeed?

This classification relates to Brna's. Applying Brna's classification onto it, we consider these descriptions as the symptom and program misbehaviour descriptions. In the debugging systems of [Shapiro 83; Lloyd 86; Edman 83; Ferrand 85; Pereira 86; Dershowitz & Lee 87], nontermination are trapped by allowing only a fixed stack size during program execution while information about goals that succeed when they should fail or goals that fail when they should succeed are specified to the debugging systems. Thus the debugging system can be seen as attempting to seek the program code error description that explains the symptoms or the program misbehaviours. For example, the diagnosis of the cause of a goal that succeeds when it should fail may be that there is error of program code in the clause

$p :- p_1, p_2, \dots, p_n.$

where there is an instance

$p' :- p'_1, p'_2, \dots, p'_n.$

such that all the p_i 's are true but p' should not be true.

Classifications of bugs are important as they raise the following questions in our attempt to build a PITS :

1. What levels and types of bug do we want an automated debugger to detect?
2. Having detected the bug(s), how can an automated debugger describe them to the student?

While we are also concerned here with the symptom description and the program misbehaviour description levels, our main interest here is at the program code error level. Our debugging approach can be divided into two parts : firstly, detecting general program

anomalies at the symptom description and the program misbehaviour description levels and highlighting them to the student; and secondly, using task-specific information to detect bugs at the program code error level. The second part of the analysis produces the location, description and fix of all bugs which will then be used by the student modelling component to infer possible misconceptions. All this information can then be used by the tutorial expert to decide how to interact with the student to debug his program and to resolve his misconceptions.

2.5 Alternative Automated Debugging Approaches

Various researchers have reviewed the literature on computer systems that use AI techniques and approaches for teaching and advising on programming. [du Boulay & Sothcott 87] classifies the systems into tutors, bug-finders and support environments depending on the educational role they play, and provides a good introductory survey of the field. [Wenger 87] provides fairly lengthy discussions of some of the systems. In Johnson's survey [Johnson 85], he contrasts other program debugging approaches with his *intention-based* approach in PROUST. Murray [Murray 86] reviews the literature by comparing other debugging methodologies with his approach in Talus which does reasoning about the computational semantics of recursive programs using a Boyer-Moore theorem prover. Our survey here emphasizes debugging methodologies. We address their limitations to motivate our proposal that a combination of different debugging approaches is suitable for Prolog programs. The approach we propose has the flavour of an intention-based system as in Johnson's sense that code analysis is based on the programmer's goals and their realisations into code.

Murray [Murray 86] gives a taxonomic classification of various debugging methodologies developed over the last fifteen years. There are two main flavours :

- dynamic analysis that examines the running of the program on specific examples, and
- code or static analysis that works on the code.

In the following sections, we briefly describe systems that exemplify different debugging approaches and examine the more recently implemented systems in greater detail.

2.5.1 Dynamic Analysis

There are five main kinds of dynamic analysis, each extracting different amounts and kinds of information from a program's execution. The simplest, used in BIP [Barr 76], is to compare program output on given test cases with expected output. The second approach examines program side-effects as well as input and output pairs. For instance, MYCROFT [Goldstein 74] analyses the side-effects of simple LOGO programs according to a propositional

description of the intended figure. The next approach records even more information about a student's program. SNIFFER [Shapiro 81] is a LISP debugger that records a complete trace of all procedure calls and changes to mutable data structures. PTP [Eisenstadt 85] looks at an exhaustive run-time trace of a Prolog execution and identifies suspicious symptom clusters that are indicative of the cause of the particular program failure. SCENT [McCalla et al 86] is a LISP program advisor whose design incorporates a blackboard system to integrate information from six levels of program analyses, ranging from the program code, the program behaviour, recognising particular patterns of code, analysing the algorithm used and its implementation into code, and student modelling. The program behaviour level includes program traces and cross-reference listings which are posted on the blackboard which can then be used by other levels of more task-specific analyses.

The fourth approach diagnoses errors using simulations of buggy models of problem-solving similar to that of [Burton 81]. [Coombs et al 86] proposes an approach to debugging user misconceptions of Prolog search behaviour. It modifies a correct Prolog interpreter to construct various *buggy interpreters*, and simulates the execution of these interpreters to find out which reproduces the user's account of execution which needs to be specified. This imposes on the user the need to provide a detailed trace of execution as he sees it, and this will be tedious and difficult for a programmer, let alone a novice. We cannot really avoid this by comparing the buggy interpreter account of execution with the correct account of execution, which need not then be specified by the user, for the following reasons :

- this assumes that the user knows what is the correct account of execution which he mistakenly believes that he can arrive at by running his buggy program,
- this assumes that any backtracking misconception is consistently manifested in the user's program.

Coombs' system tracks only six buggy rules of Prolog backtracking behaviour. As the number of possible buggy rules students can perceive seems to be more, it is restricted in handling real novice programs.

The last kind of dynamic analysis approach is qualitatively different : *interactive execution*. PDS6 [Shapiro 83], for instance, debugs pure Prolog programs by monitoring program execution. It relies on an oracle to provide answers about the correctness of program traces to detect three types of bugs :

- nontermination of a predicate invocation
- partial correctness/soundness of a program
- completeness of a program

PDS6 corrects bugs by generating and searching *bug equivalence* classes. A bug equivalence class is a set of clauses which are perturbations of a buggy Prolog clause. The perturbations are meant to correct common errors, for example, variable misspellings and the incorrect use of arithmetic tests. The buggy clause is used as the root of a search tree. A plausible clause is suggested to the oracle who decides whether to accept it. If it is not accepted, the search continues.

The limitations of PDS6 as a debugger for tutoring purposes are :

- It relies very much on an oracle. It is not always easy to answer queries about the correctness of intermediate results, in particular, for a novice learner.
- It can only debug programs based on the examples provided.
- It cannot handle the metalogical and extralogical predicates like the cut in Prolog.
- It does not reason with the semantics of program constructs. For example, it does not optimise $(X > Y, Y > X)$ to $(X = Y)$.
- Its bug correction algorithm only considers one type of bug to generate the bug equivalence class. Thus corrections for clauses with multiple bugs from differing equivalence classes cannot be discovered.

Improvements have been suggested to declarative program debugging as in PDS6 but they have not addressed these limitations. However, work on the formalising of the diagnosis of errors in pure Prolog programs continues [Edman 83; Ferrand 85; Lloyd 86]. The feature of PDS6 we utilise here in our work in APROPOS is its capability in localising the source of the program bug and explaining this to the user. Our debugging approach for Prolog makes use of Shapiro's bug diagnosis routines in PDS6 and automates the oracle by a heuristic match of the user's predicates with some reference predicates from a correct program so that an execution of the reference predicates can provide answers to the desired results of running the user's predicates.

2.5.2 Code Analysis

In static or code analysis, the code is typically transformed into some internal representation that explicitly represents control and dataflow and leaves out some details of the implementation. There are three approaches in code analysis : *structure-based* analysis, *symbolic evaluation*, and *program verification*.

2.5.2.1 Structure-Based Analysis

The first approach under code analysis is structure-based analysis. Structure-based analysis looks for surface structural forms in programs and analyses programs by accounting for

differences between observed forms and expected forms. The approach can be syntactic if it does not represent and reason about the semantics of programming language constructs. The approach cannot reason with what the program is intended to do if it does not represent and use some mapping from task goals to the internal representation of the code that are supposed to achieve the goal. To address this limitation, structure-based analysers have attempted to reconstruct this mapping, as in PROUST and in our system APROPOS2.

There are three approaches under structure-based program analysis : *plan-parsing*, *heuristic plan recognition* and *analysis-by-synthesis*. A plan is a mapping from a task goal to the internal representation of the code that achieves that goal. In the plan-parsing approach, plans are represented as a grammar and student programs are parsed in terms of the grammar. Ruth's program analyser [Ruth 76] uses a plan grammar which is a representation of the program task algorithm to attempt to parse the student's program. Bugs are indicated by a failed parse. SPADE [Miller 79], the successor of MYCROFT, is a plan-based programming editor that uses a context-free grammar of planning and debugging routines to parse and understand the problem-solving process of the user.

In the heuristic plan recognition approach, plans are retrieved from a plan library and partially matched to student code. LAURA [Adam & Laurent 80] constructs a representation of the control flow of the program and matches against that of a correct program. Differences in matching are treated as bugs. MENO-II [Soloway et 83] specialises in the analysis of iterative loops and related variables like control variables or counters in Pascal. It parses the student code into a parse tree which is matched against a plan template. MENO-II using a simple template match was tested empirically and failed to diagnose a large portion of programs.

PROUST [Johnson 85] was subsequently developed to address some of the limitations of MENO-II. It decomposes

- the task specification into a set of goals, and
- for each goal, the possible plans which could be selected, and finally,
- the different ways in which plans can match code.

PROUST matches more than one plan template to a single program, each template corresponding to a task goal. PROUST uses a program library that associates task goals with plan templates which are used to match against a student's program. Matching is done by associating plan templates with expected code in the student's program. A plan can specify subgoals which are to be added to PROUST's agenda of plans. The intentions of the program statements are considered determined when each statement has been matched to some part of a plan and there are no unaccomplished goals on the goal agenda. Thus PROUST's approach

is also one of analysis by synthesis. Since an exhaustive search is not practical, heuristics are used to select one plan that predicts the fewest bugs in the student's code. Plan-difference rules are used to explain any discrepancy between the student code and the code predicted by matched plan templates. They either explain the discrepancy as common implementation variants or as specific program bugs.

The limitations of PROUST as a debugger for tutoring purposes are :

- It can fail in its analysis, or give a false alarm, or fail to detect unrecognised bugs due to failure of plan-recognition heuristics. For example, the cheapest prediction may not be the most appropriate, resulting in selection of an incorrect plan.
- A plan-difference rule could be present to explain a discrepancy, but may fail to fire if its triggering conditions are incorrect or insufficient. The set of rules are written based on extensive empirical studies of Pascal programs written by students.
- It cannot analyse programs which are far removed from the task specification.
- It does not reason with the semantics of program constructs.
- It can only analyse a limited range of Pascal constructs.
- The approach cannot be easily generalised to beyond the RAINFALL and BANK kind of tasks where the goals are fairly constrained and where it is unlikely the student will have additional or different goals in mind.

Talus [Murray 86] can detect certain types of bugs in its code-matching algorithm recognition phase, but its main approach to detecting bugs is based on program verification. As described earlier, SCENT has six levels of program analyses corresponding to the conceptual levels of the program design and the programmer's knowledge. One level is the analysis of what algorithm might be used and the implementation of it using a library of different tasks and the different algorithms for each task. A bug finder at the implementation level has been implemented [Huang 87] in which discrepancies are first explained by a library of common correct and incorrect variants of the code, failing which a theorem prover (which is not yet implemented) is to be invoked to try to establish code equivalence. If code equivalence cannot be established, the discrepancies are explained as bugs.

In the analysis-by-synthesis approach, plans are synthesized into code. The debugger can synthesize complete solutions to solve a program task. GREATERP [Reiser et al 85] is an interactive LISP tutor that relies on

- a library of production rules to synthesize correct code that satisfies the task specification, and
- a library of buggy rules to synthesize code that might match a student's incorrect code.

Associated with each of these buggy rules is a template error message. GREATERP tries to follow the student's design decisions as the student develops the program. As each token in the student's program is entered, GREATERP checks to see if the student is following a design path known to be correct or buggy. Buggy paths are disallowed as soon as they are detected by explaining the error to the student and then allowing the student to try again. GREATERP can provide the code if necessary. The limitations of GREATERP for tutoring purposes are :

- The tutoring approach is constrained to be behaviourist. The student is highly constrained in the solutions that can be developed. He is also constrained to a top-down view of program design. As Anderson [Anderson et al 87] himself notes, the important thing for the student to do for learning is to go through the thought process that generates the answer rather than copy the answer from the feedback.
- If two design paths lead to alternative implementations and they coincide at some point, GREATERP cannot determine which design is being followed until the implementation is further elaborated. Since it cannot defer judgement, it does not allow these kinds of variability and selects one path to follow.

In recognition of these limitations, Anderson and his co-researchers are now exploring with a new PUPS-based tutor that gives feedback after complete LISP expressions like (APPEND (LIST X) Y) instead of after each token.

In Prolog where there are fewer system predicates to anchor the analysis, more work needs to be done to study representations of Prolog programming techniques or plans if we are to attempt a production rule approach for synthesizing Prolog programs.

2.5.2.2 Symbolic Evaluation

The second approach under code analysis is symbolic evaluation. The symbolic evaluation approach is to determine which paths through a program are executed using symbolic inputs. It differs from *symbolic execution* in the production of symbolic expressions for program outputs by static analysis rather than by dynamic interpretation [Cheatham et al 79]. PUDSY [Lukey 80] analyses a Pascal program by evaluating the program symbolically to derive a formula that relates outputs to inputs, recording which part of the program is responsible for which part of the formula. It compares the formula against a model formula for the task and identifies parts of the program responsible for the differences. However, the problem of checking and proving formula equivalence is a difficult one. PHENERATTE [Wertz 82] evaluates each of the possible paths of a LISP program and uses a set of rules describing general program constructs and typical methods of coding to repair anomalies like non-terminating recursion.

2.5.2.3 Program Verification

The third approach under code analysis is based on formal verification. Formal proof uses mathematical theorem-proving techniques to verify that assertions, embedded in the program code, are valid when compared to the verification conditions that are generated from traversing the different possible program paths. Assertions are usually represented in some predicate calculus form, inserted by the programmer, which describe formally the specification of the task. They describe pre-conditions, invariant conditions and post conditions of a segment of code.

The formal approach of [Katz & Manna 76] is to synthesize inductive invariants from program statements and using these invariants to establish a proof of correctness. For error correction, program statements are altered so that the necessary inductive invariants are derived. However, the problem of synthesizing inductive invariants and the problem of determining what problem statement to correct from a typically long and tedious proof are very difficult. Indeed no implementation using this approach exists to date.

Talus [Murray 86] debugs recursive LISP programs by repairing induction proofs in the Boyer-Moore Logic. It differs from all debuggers in that it reasons about computational semantics to detect and correct bugs in students' programs. It compares the student program to a model program known to be correct. Talus performs a heuristic best-first search to choose between competing algorithms and to map student functions to reference functions. Heuristics are used to map student variables to reference variables. For each function pair, TALUS divides them into conditional cases and for each pair of cases of the student's function and the reference's function, tries to prove functional equivalence between the action parts of each pair. If the student code differs from the reference code, an induction proof of equivalence is constructed. This generates verification conditions in the Boyer-Moore Logic. Failed verification conditions are interpreted as indicating bugs in the student's program. These bugs are corrected with code fragments from the reference program. The limitations of Talus are :

- The power of a formal approach of program verification only comes after the successful use of heuristics in algorithm recognition, function and variable mappings which can go astray. For example, the way inductive proof is applied assumes the correctness of the test parts of the conditional cases of the student programs. If the test part is incorrect, TALUS may not be able to match its conditional case to a case in the reference program or if it can, it may incorrectly fix the action part of the student conditional case instead of the test part.
- The program verification component using the Boyer-Moore theorem prover may computationally increase as the size of the program grows.

In APROPOS2, we developed a similar heuristic best-first search to choose between competing algorithms and implementations of algorithms, and to map student predicates to reference predicates.

2.5.3 What Debugging Approach for Prolog?

We have discussed various debugging methodologies and described specific limitations of programs implementing these methodologies. What methodology is practical for debugging Prolog programs? We propose an eclectic approach which combines elements of some of these methodologies. In addition, we propose the use of multiple sources of program analysis to help in the debugging. Static analysis for Prolog appears to offer various advantages and disadvantages. On the positive side, obvious slips and miscodings can be spotted quickly. Unreachable and redundant code can be detected and analysed. By examining the context surrounding a bug, misconceptions can be inferred and corrections to the bug suggested. No special handling of side-effects is required although they still need to be reasoned about. Programming style can be detected as well as tutored. On the negative side, there is a need to interpret the student's intentions through heuristic code-matching, but that can be cumbersome and is dependent on having a representative library of plans and bugs.

Dynamic analysis also has its pros and cons. It can cope with unforeseen user strategies which is useful for recognising unexpected correct ones. It can find errors without doing a full analysis. It can more readily find out any bug symptoms and program misbehaviours. On the other hand, unreachable and redundant code cannot be detected. An oracle to answer queries about the desired results of program execution is needed. Programs are only debugged with respect to the examples tried. Debugging impure Prolog programs (those that use the cut, for example) and programs with side-effects is difficult since execution traces must be analysed together with program states.

In order to harness the advantages of both static and dynamic analyses, we propose a combination of debugging approaches for Prolog. The approach is primarily heuristic code-matching with dynamic analysis supplementing code-matching when the latter fails for parts of the analysis. A static analysis permits an automated debugger to debug bugs and suggest corrections for bugs as there is some reference to be matched against. With dynamic analysis, an automated debugger is able to collect all the solutions a student program produces on successive forced backtracking from a goal call, and compare and analyse these solutions vis-a-vis the correct solutions. Dynamic analysis allows an automated debugger to explore run-time properties and behaviour of Prolog programs at a lower computational cost than an often quite involved static analysis. It is used also to confirm the run-time behaviour and properties arrived at by static analysis, which in turn can be used to explain the program's run-time behaviour.

In the following chapters, we will discuss the roles static and dynamic analysis play in the debugging of Prolog programs in our system APROPOS2, and describe how programming and task knowledge structures are used in APROPOS2.

Chapter 3. An Overview of APROPOS2

In this chapter, we provide an overview of the debugging process of APROPOS2 and will present a case study. APROPOS2 will examine a program that has no syntax errors picked up by the Prolog interpreter when it consults the clauses. APROPOS2's analysis can be decomposed into two parts. The first part performs multiple program analyses on the programs. The aim is to seek a proficiency comparable to that of a quick scan of the program by an experienced Prolog programmer who has no knowledge of what the program is intended to do. The second part uses task-specific knowledge to analyse the program, detect bugs and correct them. The aim is to achieve a proficiency comparable to that of a Prolog instructor who having set the task now evaluates and comments on the program. However, this human metaphor for an automated debugger is not required to be strict, for an automated debugger can perform program analyses which are impossible or difficult for the human debugger, and vice versa.

Figure 1-3 shows an overview of the debugging process in APROPOS2. The first part of the analysis requires multiple program analyses. Figure 3-1 shows the different types of program analyses which detect program anomalies and generate program information for the second part of the analysis. Multiple program analyses entail a multi-pass analysis of the program source code.

3.1 Multiple Sources of Program Analyses

In this section, we will develop in greater detail the argument that mode analysis, type analysis, dataflow analysis, mistyping checking, program transformations, and plan-fitting or technique-fitting all have a part to play in analysing novice Prolog programs and that these sources of analyses help each other. Our main use of the information derived from these sources of analysis is to guide the code-matching process used in APROPOS2 which will be described in Chapter 5.

We discuss these forms of program analyses. For each form of analysis, we describe the methods used, and how we have incorporated and used them in APROPOS2. In Chapter 8, we propose how we can further extend their usage in analysing Prolog programs for debugging and tutoring purposes. We will not delve into the theoretical frameworks of some of the analyses for which there is a wider body of literature from the logic programming community.

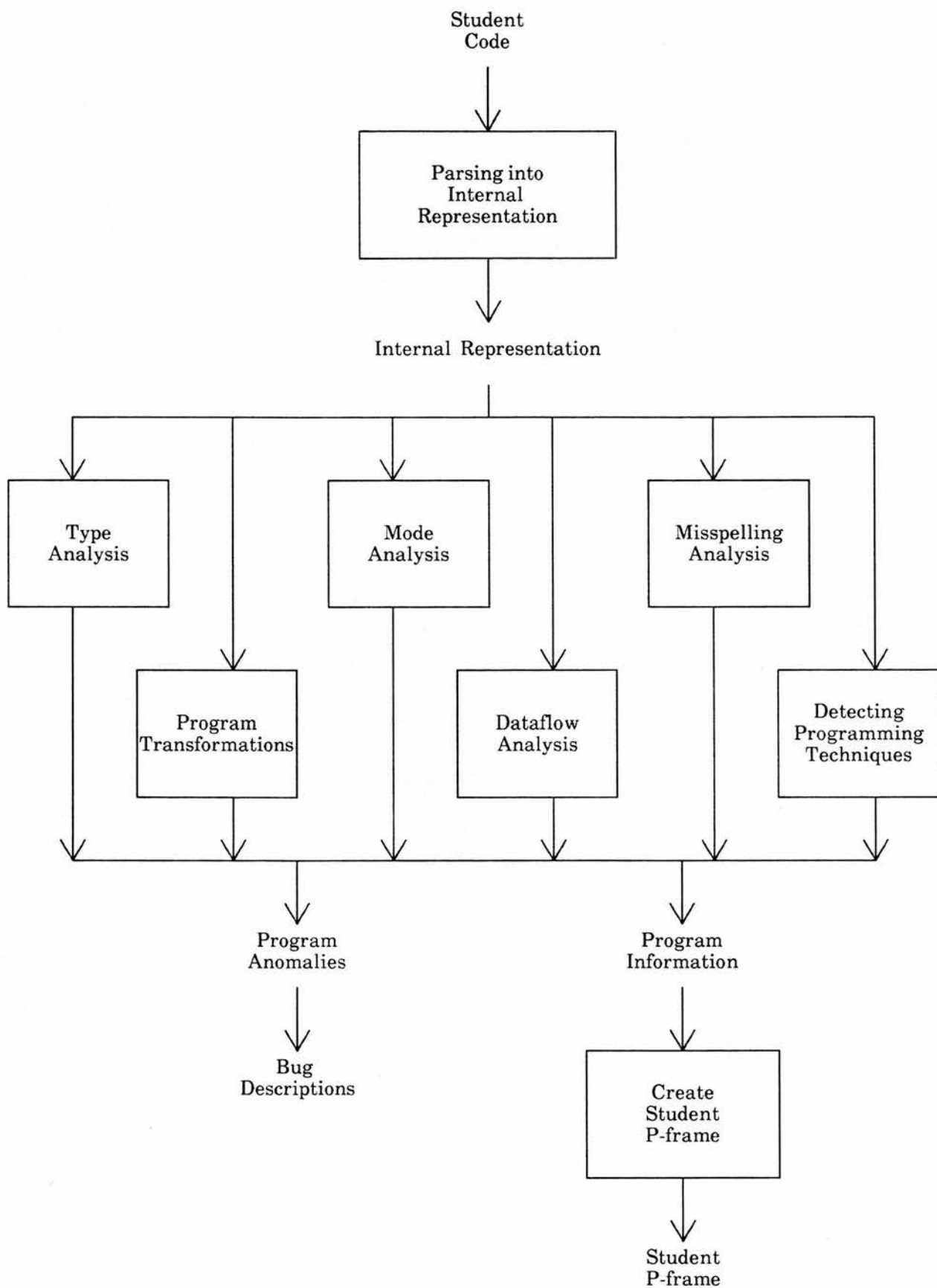


Figure 3-1 : Multiple program analyses in APROPOS2

3.1.1 Mode Analysis

In this section we describe how we have made use of mode analysis to reason about Prolog programs in APROPOS2. The importance of bringing mode information to bear on the analysis of Prolog programs stems from the observation that in practice Prolog programs can only be used in certain mode invocations to compute different input and output relations.

3.1.1.2 Inferring Modes

Mode inference is used to determine which arguments of a predicate are used as input parameters in a Prolog program and which are used as output parameters. In APROPOS2, the problem specification states the mode declaration of the main predicate for which the student is asked to write a predicate definition. It is reasonable to do this, as APROPOS2 is intended for novices in their initial learning phase. If the student program includes auxiliary predicate definitions, then we need some way of inferring the invocation mode and the supportive mode of these predicates which will be used to analyse their predicate definitions.

Previous work in inferring invocation modes from Prolog programs have been done by Mellish [Mellish 81] and Debray [Debray 85; Debray & Warren 86] to do compiler optimisation. The method of Mellish proceeds in two parts. The first part examines all the places in the body of clauses where a goal for the predicate can be invoked and goes through the whole program to determine how the possible values of a variable at different positions depend on one another. It produces a network of dependency links. The second part propagates constraints through this network to establish modes for all the argument positions of a predicate. Debray's method involves the derivation of a mode inference program customized to the Prolog program being analysed and then the execution of this mode inference program to get the modes. Lever [Lever 85] proposes a method of inferring the output instantiation state of a predicate invocation after a successful call given its input instantiation state before the call.

Our interest here differs from that of Mellish, Debray and Lever, as we focus our attention on the use of mode information for debugging Prolog programs. We use Mellish's method in APROPOS2 to infer a set of input modes for each predicate after examining the whole program. Each input mode of a predicate *summarises all the ways* they may be called by running the program in various ways.

However, the methods therefore cannot be used to detect whether predicate calls that occur in the body of clauses have invalid argument modes, that is, modes which are not a subset of the supportive input modes of the relevant predicate definitions. To illustrate, Mellish's program would infer a mode of `insert(+, ?, ?)` from this program :

```

sorting([X|T], L) :-
    insert(X, T1, L),
    sorting(T, T1).
sorting([], [X]).

insert(X, [H|T], L) :-
    X < H,
    append([X], [H|T], L).
insert(X, [H|T], L) :-
    X > H,
    append([H], L1, L),
    insert(X, T, L1).
insert(X, [], [X]).

append(L, [], L).
append([H|Y], L1, [H|Z]) :-
    append(Y, L1, Z).

```

and from information that `sorting/2` is to be used for mode `sorting(+, -)`. Whereas a supportive mode of `insert(+, +, ?)` can be inferred from the `insert/3` predicate definition itself and that the `insert` subgoal invocation in `sorting/2` is of mode `insert(+, -, -)` which therefore cannot be handled by this `insert/3` predicate definition.

We need to infer supportive modes on a local basis for the head predicate of predicate definitions, based only on information in these predicate definitions and the known supportive modes of system predicates. We can then check that subgoals in other clauses that invoke this predicate do so in a mode that can be handled by the inferred supportive mode of the predicate. The difficulty with inferring supportive modes using local information is that it may be the case that not enough mode information is available to help infer modes. In Chapter 8, we propose some ideas on inferring how a predicate definition may not be used for certain modes as its invocation may result in nontermination.

3.1.1.2 Uses of Mode Information

The mode information inferred and known about predicates can be used for these purposes :

- to help in heuristic code-matching. In APROPOS2, we use a heuristic code-matching approach to estimate how well a student's program matches a reference program so as to select one reference program which best fits the student's. One measure of the closeness of fit is the mode information, thus two predicates are better matched if the modes of one is subsumed in the modes of the other than if otherwise.
- to know which argument positions or variables to instantiate to test cases for dynamic analysis to be run on a predicate using test cases. To illustrate a simple example, suppose for the `sorting/2` problem, APROPOS2 is matching this program clause :

```

sorting(List, Res) :-
    append(X, [A,B|Y], List),
    A > B,
    append(X, [B,A|Y], SwappedList),
    sorting(SwappedList, Res).

```

against this reference clause :

```

sorting(List, Res) :-
    swap(List, SwappedList),
    sorting(SwappedList, Res).

```

APROPOS2 will find a code mismatch between

```

append(X, [A,B|Y], List),
A > B,
append(X, [B,A|Y], SwappedList)

```

and

```

swap(List, SwappedList)

```

and will then try to show their equivalence by running them on test inputs. Since the first `append/3` subgoal invocation has a mode of `append(-, -, +)`, APROPOS2 will execute it with the third argument instantiated to some test datum.

- to help recognise what programming technique might be used in the program. To illustrate, one of the techniques APROPOS2 recognises is the use of the same variable in the same argument position of a predicate to pass back an accumulated result built up in other argument positions. One of the inputs of this analysis is the mode declaration of the predicate under examination, for APROPOS2 will only find such variables if they occur in a '-' or '?' mode. APROPOS2 will detect this technique in

```

reverse([], L, L).
reverse([A|X], Y, Z) :-
    reverse(X, [A|Y], Z).

```

if this `reverse/3` is known or inferred to have modes `(+, +, -)`. Further analysis will detect the iteration argument from a '+' position and the accumulator argument from a '-' position. What use can we make of this information obtained? We know now what the iteration, accumulator and result arguments are in the program, so we can compare them against the iteration, accumulator and result arguments of a correct reference program. This approach is used in our first version of APROPOS, APROPOS1. In APROPOS2, in which we use a primarily code-matching approach, we use a score to measure how well a student's program matches a reference program in order to choose which reference program best matches the student's. One score measure is the use of programming techniques, thus the use of identical techniques in both programs contributes to a better score, while the use of different techniques worsens the score.

- to suggest bugs or bug behaviour in the program, for example, uninstantiated variables that occur in arithmetic operations, or system predicates that would fail at run-time due to insufficient instantiation. This use of modes is currently not yet incorporated into APROPOS2.

3.1.2 Dataflow Analysis

Dataflow analysis as a method for analysing internal program behaviour has been used originally in global program optimisation and the study of advanced computer architecture. In dataflow analysis, the program is scanned in a systematic way. Information about the use of variables is collected so that certain inferences can be made about the effect of these uses at other points of the program. It does not mean execution of the program being analysed.

When used to detect programming errors, dataflow analysis identifies *anomalies* in the pattern of definition and use of variables. It detects [Johnson 85] :

- (1) references to undefined variables,
- (2) assignments of variables of values which are never used,
- (3) unreachable code in the program,
- (4) infinite loops.

The presence of a dataflow anomaly does not mean that the execution of the program will definitely produce incorrect results. It means only that execution may produce incorrect results. Producing correct or incorrect results may depend on the input data. The presence of a dataflow anomaly is at least a cause for concern because it often is a symptom of an error.

Common programming errors such as mistyping and omission of clauses in Prolog suggest dataflow anomalies. Such errors may arise from misconceptions about Prolog. To check an anomaly like that of (1) for Prolog would require mode analysis. To check an anomaly like that of (2) for Prolog (instantiations of terms to values which are never used) would probably make less sense or is at least difficult since all arguments are in a sense implicitly used for and affect matching. To check for infinite loops in Prolog would mean looking for anomalies in code like

```

functor(..., [], ...).
functor(..., T, ...) :-
    functor(..., [H|T], ...).

```

which is likely to cause an infinite loop to arise from execution of the recursive clause.



Our use of dataflow analysis is for detecting dataflow anomalies. A common form of dataflow anomaly observed in novice Prolog programs is where one argument position has a term which keeps building up resulting in a nonterminating computation or an incorrect computation because that argument can never match the structure in the corresponding argument in the base case. APROPOS2 checks for this. A dataflow analysis is done to check if arguments of successive recursive calls descend along some well-founded ordering. If the ordering is violated, then APROPOS2 will warn of the potential for looping. There has been other work done on detecting loops in Prolog both at compile-time and at run-time [Covington 85; Brough & Hogger 87]. Here is an example of APROPOS2's output for the task *delete(+Thing, +FromThisList, -GetThisResult)* which deletes occurrences of the term *Thing* from the list *FromThisList* to get the result list *GetThisResult* :

Well, these clauses ...

```
delete(E, [], []).
delete(E, [E|T], R) :-
    delete(E, T, R).
delete(E, [H|T], R) :-
    delete(E, T, [H|R]).
```

Dataflow error in argument 3 of delete/3.

This gives rise to incorrect computation as this argument will build up into a larger list during recursive calls. The resulting recursive call will never match the base case.

This also illustrates the problem of giving an apposite explanation. Our intention is that APROPOS2's analysis will be part of the input to a tutoring system whose job it will be to create suitable explanations.

APROPOS2 uses a program written by O'Keefe [O'Keefe 84] to check for missing base cases to detect nonterminating computation. It converts all predicates to propositions with no arguments and then tries to prove each different user-defined proposition. This fails if base cases are present but unmatchable. Here is an example of APROPOS2's output :

Your program clauses are :

```
reverse([], Y).
reverse([], []).
reverse(A, B) :-
    recur(A, B).
recur([H|T], Y) :-
    recur(T, [H|Y]).
```

recur/2 has no terminating proof. Either there is a missing base case or it calls other predicates that have no terminating proof themselves.

3.1.3 Type Analysis

In languages like Pascal and Ada, the types of variables and function names are defined by redundant declarations, and the compiler checks the consistency of definition and use. In a language like Prolog, explicit declarations are not required to be specified, but we can still infer the type of expressions from local context, while still establishing consistent usage. Types in Prolog represent sets of terms. Type inference can be used to infer the types of expressions when little or no information is given explicitly. We can assign types to all possible calls in which a predicate can be used. The type of every term may be inferred from the context using the type information of terms known. In general, the type inference problem for Prolog is undecidable. Type inference reduces to type checking when there is enough type information in a program that the type inference task becomes trivial.

3.1.3.1 Inferring Types

There are two main approaches to types in Prolog. In the Mycroft-O'Keefe system, a type assignment for a program is considered valid if it is preserved under resolution ("well-typed programs do not go wrong") [Mycroft & O'Keefe 84]. This approach deals mainly with type checking. In Mishra's approach, the type of a predicate is characterised as a conservative description of its success set ("ill-typed programs cannot succeed") [Mishra 84]. This approach deals mainly with automatic type inference. Both systems can handle polymorphism, that is, permit type variables in the typing of a term.

Our approach in APROPOS2 is more of type inference, similar to Mishra's approach. The type declaration of the main task predicate is specified to APROPOS2. For each student auxiliary predicate definition, APROPOS2 infers the supportive types of the head predicate in the definition for which it may be used. APROPOS2 then checks if whenever these predicates have been called as subgoals, the arguments of these subgoals' invocations are of the inferred or known supportive types. The type inference method deduces types of variables from the known type declarations of predicates, including system predicates for which the type declarations are also known, and propagates this information within each clause to infer unknown types of predicates. This is repeated until no more type information can be further deduced.

Currently, our method only handles a few types, namely, atoms, numbers, lists of atoms, lists of numbers, lists of the previous types, and difference lists. More elaborate methods of type inference have been proposed [see for example Kanamori & Horiuchi 85; Debray 86; Yardeni & Shapiro 87] and can serve as better alternatives to our simple type inference program. To illustrate an example of APROPOS2's output, from information that ' $</2$ needs arguments to be of type number, APROPOS2 will infer from this insert/3 predicate definition :

```

insert([], X, [X]).
insert([Y|Ys], X, [Y|Zs]) :-
    X < Y,
    insert(Ys, X, Zs).
insert([Y|Ys], X, [X,Y|Zs]) :-
    X >= Y.

```

that it works only for invocations with the types *insert(list(number), number, list(number))*. This information will be used in further analysis, for example, to address the discrepancy if the student's sorting/2 program is :

```

sorting([X|Xs], Ys) :-
    sorting(Xs, Zs),
    insert(X, Xs, Ys).
sorting([], []).

```

where the insert subgoal call has types *insert(number, list(number), list(number))*. However, a spelling mistake can easily mislead type analysis.

APROPOS2 will warn of the use of a possibly unintended list structure [H|T] if the program clause contains terms of the form [H|T] and there is no other type information that tells that T is a list. Here is an example of APROPOS2's output :

```

Warning. In the clause
    reverse([Head1|Tail1], [Something|Head1]) :-
        reverse(Tail1, Something).
if Head1 in [Something|Head1] is not a list, perhaps an atom, then [Something|Head1]
may not be what you want.
A note about syntax : [a|[b]] is equivalent to [a,b], and [a|[b,c]] is equivalent to [a,b,c] but
[a|b] is not the same as [a,b].

```

3.1.3.2 Uses of Type Information

Type information is used for these purposes :

- to help in heuristic code-matching. In the second part of APROPOS2's analysis which uses task-specific information, we use a scoring function to measure how well a student's program matches against a reference program. One measure that contributes to the scoring function is the types specified or inferred of the predicates.
- to know what types of terms to instantiate to argument positions for dynamic analysis to be run on a predicate using test cases. To illustrate, APROPOS2 will infer from this program of list-reversal which requires the types of both arguments of reverse/2 to be lists :

```

reverse([], []).
reverse([A|X], Res) :-
    attach(Sofar, [A], Res),
    reverse(X, Sofar).

```

that attach/3 needs arguments to be all of list types since Sofar, [A] and Res are all of type list.

- to verify that the types of the result arguments of a goal call which has succeeded are in agreement with the declared and expected types. To use the same example of the reverse/2 program, if attach/3 is inferred to use mode attach(-, +, -), we want for a goal call of attach(-, +, -) on the student's attach/3 program to validate that the types of the output arguments, when instantiated on a successful goal call, will be of list types.
- to help to confirm or reject misspelling hypotheses. Currently, we have not incorporated this use of types into APROPOS2.

3.1.4 Misspelling Analysis

Prolog programs are very prone to misspelling errors as such errors do not result in compile-time errors but in programs with different, unintended meanings. APROPOS2 detects instances of singleton variables, and instances of possible mistyping mistakes of variable names. APROPOS2 also checks for references to undefined predicates. Several Prolog systems now provide a static cross-referencing tool at consult time for identifying defined predicates not called or missing predicate definitions. Several versions of Prolog already provide single variable checking and non-contiguous clause checking.

The main difficulty with using a spelling checker for Prolog programs is that words that would be detected as spelling aberrants of another word may not be spelling mistakes at all. For example, it is common to use variable names like L, L1, L2 and L3 in a Prolog program. However, we can still go some way by heuristically postulating mistyping errors through utilising other information such as the whole Prolog program, and by using modes and types information.

Misspelling of predicate names and of variable and atom names can be treated in different ways. A first scan of the student program is done to derive a list of all predicate names used, with associated information like possible modes and types of the argument positions. This will serve as the dictionary of words against which hypotheses are made to suggest spelling errors. Heuristics can be used to guide the spelling checker, for example :

- invoke the spelling checker for comparing two functors if they have the same arity, and the mode and type declarations of one are identical or subsumed in another.
- check whether missing predicate definitions can be explained by mistyping mistakes.
- check whether the clauses for a particular predicate are not contiguous within the input file. If so, the intrusive clause may involve a typing mistake.

Spelling errors of variable names especially short ones are more difficult to detect. Inappropriate singleton variables are the likeliest candidates. The spelling checker in APROPOS2 only checks for possible misspelling of variable names if they are at least 3 characters long. For each singleton variable in a clause, it checks if the variable name might be a spelling aberrant of one of the other variable names within the clause. We classify the result of the checking into these types : no match, transposition of two characters, a different character, an extra character, and a missing character. Here is an example of APROPOS2's output :

```
!!! Warning : In the clause
               sorting([X|Tail], A1-Z2) :-
                   split(X, Tail, Small, Big),
                   sorting(Smail, A1-[X|A2]),
                   sorting(Big, A2-Z2).
these variables occur once only : Small Smail.
Check if you have made any mistyping mistake. Recall that if a variable appears in a
clause once only, we do not have to invent a name for it. We can use the so-called
anonymous variable, which is written as a single underscore character "_".
Is Smail a mistyping for Small?
```

More information may be deduced from further analysis. For example, the PRESET system of Takahashi and Shibayama [Takahashi & Shibayama 85] matches each subgoal in a clause with the head of any clause to infer irreducible subgoals.

3.1.5 Program Transformations

Program transformations have a part to play in debugging Prolog programs for tutoring purposes as they enable different unfolded forms of a normalised representation to be recognised as equivalent programs. APROPOS2 performs program simplification to facilitate code-matching with a normalised representation of a program. It applies common folding and unfolding rules to the student's program and the transformed program can then be used for subsequent analysis.

Currently, the rules incorporated in APROPOS2 are very specific. APROPOS2's code-matching process maps student predicates to reference predicates. An example of a folding rule is to fold occurrences of *functor*([H], T, Res) that

- has been matched to the reference predicate of *append*/3 and that
- appear in a program clause (before a cut) which has only a conjunction of subgoals in its body

by replacing occurrences of Res in the clause by [H|T]. Another folding rule is to fold occurrences of Res=[H|T] that does not appear after a *var*(Res) subgoal within the body of a clause. Here is an example of APROPOS2's output :

```

Your clause :
    reverse(X, Y) :-
        X=[H|T],
        New = [H|Y],
        reverse(T, New).
can be folded into
    reverse([H|T], Y) :-
        reverse(T, [H|Y]).

```

Devising program transformation rules and suitable conditions for their invocation turns out to be more subtle than we have expected. Some of the folding rules we had initially incorporated into APROPOS2 are incorrect. For example, consider this task :

Task : atom_count/2

Write a Prolog program *atom_count*/2 which accepts a list as first argument and instantiates the variable given as second argument to the number of atoms in the list. The built-in predicate *atom*/1 succeeds if and only if its argument is an atom. Example :

```
:- atom_count([a, a, [b,a], []], Res).
```

Res= 3

% ie. the first two "a"s and the [] (which is an atom, as well as a list).

The procedural semantics of this student program which is correct will be altered by folding the subgoal "N is Numberlist" :

```

atom_count(List, N) :-
    atomcount(List, Numberlist),
    N is Numberlist.
atomcount([], 0).
atomcount([Head|Tail], Number) :-
    atomcount(Tail, Number2),
    atomincrement(Head, Number, Number2).
atomincrement(Data, N + 1, N) :-
    atom(Data).
atomincrement(Data, N, N).

```

We will discuss APROPOS2's analysis of this student's program in Section 7.3.

To illustrate another example, to avoid folding occurrences of $\text{Ans} = [\text{H}|\text{Ans}]$, we need to incorporate an occurs check. The conditions for applying folding and unfolding rules to a program so as not to affect the semantics of a program have to rely in part on available mode information. Consider this `append/3` program :

```
append([], L, L).
append([H], L, [H|L]) .
append([H|T], L, Ans) :-
    append(T, L, X),
    append([H], X, Ans).
```

By folding `append([H], L, [H|L])` into the recursive clause, we get :

```
append([], L, L).
append([H|T], L, [H|X]) :-
    append(T, L, X).
```

These two programs are only identical for the usage of `append(+, +, +)` or for the first solution returned. For while the second `append/3` program returns one solution only for the common usage of `append(+, +, -)`, the first `append/3` program gives the same answer infinitely many times for `append(+, +, -)`.

[Ross 87] provides another example why program transformation needs to rely on mode information. Consider this version of `reverse` which works only for mode `reverse(?, -)` :

```
reverse([], []).
reverse([H|T], L) :-
    var(L),
    ... etc ...
```

Suppose this `reverse/2` is used for the problem of checking whether a given list is a palindrome :

```
palindrome(L) :-
    reverse(L, L1),
    L1 = L.
```

This cannot be folded into :

```
palindrome(L) :-
    reverse(L, L).
```

which will fail since `reverse(+, +)` will not work properly.

3.1.6 Detecting Programming Techniques

In the heuristic code-matching approach of APROPOS2, we use a score to measure how well a representation of a student's program matches against a representation of a reference program. One measure which contributes to the scoring of the match is the use of programming techniques in both programs. The idea is that the use of identical techniques in both programs contributes to a better score, while the use of different techniques penalises the match.

One of the techniques APROPOS2 recognises is the use of the same variable in the same argument position of a predicate to pass back an accumulated result built up in other argument positions. One of the inputs of this analysis is the mode declaration of the predicate under examination, for APROPOS2 will only find such variables if they occur in a '-' or '?' mode. APROPOS2 will detect this technique in

```
reverse([], L, L).  
reverse([A|X], Y, Z) :-  
    reverse(X, [A|Y], Z).
```

if this reverse/3 is known or inferred to have invocation mode(+, +, -). Further analysis will detect the iteration argument from a '+' position and the accumulator argument from a '-' position.

APROPOS2 recognises the iteration argument by checking if an argument term in the recursive subgoal in the recursive clauses(s) is contained in the corresponding argument term in the head goal. If the argument is a '+' mode, then the argument is being destroyed recursively, as in this clause :

```
reverse([H|T], Answer) :-  
    reverse(T, ReversedT),  
    append(ReversedT, [H], Answer).
```

If the argument is a '-' mode, then the argument is being constructed recursively back up in the head goal, as in this clause :

```
reverse(List, [LastElement|ReversedAllButLast]) :-  
    getlast(List, AllButLast, LastElement),  
    reverse(AllButLast, ReversedAllButLast).
```

APROPOS2 confirms the output argument which has a known or inferred '-' mode by checking if the variable term in that argument position in the recursive subgoal in the recursive clauses(s) is a different variable from the corresponding variable term in the head goal.

The *failure-driven loop* is a technique in which the last subgoal is a fail/0 predicate which is used to force backtracking to earlier choice-points and thus attempt to resatisfy subgoals. Lynch [86b] describes an analyser that debugs simple failure-driven loops by detecting expected features of such a loop for a specific problem. It looks for instances of resatisfiable subgoals, subgoals with side-effects, and subgoals that appear after the fail subgoal, and then comments upon their improper usage by reference to its model of a loop plan of the form :

```

loop :-
    { resatisfiable goal(s) },
    { side-effect goal(s) },
    fail.
loop.

```

Detecting features of a failure-driven loop is a good starting point but a more robust analysis of failure-driven loops, considering that there can be many different ways of implementing them, requires more involved analysis such as that of dataflow. As programming tasks, whose realisations involve failure-driven loops, have task-specific goals, we want to explore the use of an analysis-by-synthesis approach to analyse how task goals may have been implemented in the student's program. The merit of using an analysis-by-synthesis approach is that by dynamically selecting, elaborating and reformulating task goals to predict in the student's program, an automated analyser can know the code realisation of what goal to look for in the student's code and thus can tailor the explanation of bugs accordingly to the student's goals. We discuss the use of an analysis-by-synthesis approach to analyse failure-driven loops in greater detail in Chapter 6.

3.2 The Task-Specific Debugging Capabilities of APROPOS2

The second part of the analysis of APROPOS2 uses task-specific knowledge to detect many different kinds of nonsyntactic bugs at three levels of abstraction :

- (1) The Algorithm Level : the design of the task solution.
- (2) The Predicate Definition Level : the predicate definition decomposition of the chosen algorithm.
- (3) The Implementation Level : the code that implements each predicate definition.

3.2.1 The Algorithm Level

An algorithm is a particular way of solving a problem that specifies a strategy for the problem's solution but leaves out details of the implementation. For example, *reverse/2* has at least 3 algorithms : the *naive* method, the *railway-shunt* method using an accumulator, and the *inverse naive* method which recursively extracts the last element of the list and reverses all but the last element and splices the extracted element back into the right place. Appendix

I.1 gives brief descriptions of the algorithms for doing list-reversal. Bugs at the algorithm level reflect lack of understanding of the task, or an incorrect or inefficient strategy for solving the task. Currently, APROPOS2 has representations of implementations of correct algorithms as well as inefficient algorithms. APROPOS2 can recognise and comment on algorithms that are correct but inefficient. We intend to incorporate implementations of buggy algorithms into APROPOS2's task representations so that it can detect them and explain why they are faulty.

3.2.2 The Predicate Definition Level

Three types of bugs at the predicate definition level are detected : missing, extra and incorrect predicate definitions. Missing predicate definitions are necessary to the implementation of an algorithm that are not present in the student's solution. For example, the student is expected to write `reverse/2` but instead has provided just a `reverse/3` definition, or the partitioning subgoal or its equivalent and its predicate definition are missing from a program implementing quicksort. Extra predicate definitions are superfluous procedures not necessary to the implementation of the chosen algorithm. Incorrect predicate definitions are procedures that have bugs at the implementation level.

3.2.3 The Implementation Level

At the implementation level, common bugs like these are detected at the coding level :

- missing clauses
- extra clauses
- missing subgoals
- extra subgoals
- wrong subgoals
- wrong clause order
- wrong subgoal order

Wrong clause order and wrong subgoal order can affect the correctness or efficiency of the implementation of an algorithm. To illustrate incorrect and inefficient implementations of an algorithm, consider the typical naive program for `reverse(+, -)` which is of order $O(n^2)$ in the number of goal invocations in computing the first solution, where n is the length of the list :

```
reverse([], []).
reverse([Head|Tail], Result) :-
    reverse(Tail, ReversedTail),
    append(ReversedTail, [Head], Result).
```

Reversing the subgoal ordering in the recursive clause will result in an *inefficient* reverse(+, -) which is of order $O(n^3)$:

```
reverse([], []).
reverse([Head|Tail], Result) :-
    append(ReversedTail, [Head], Result),
    reverse(Tail, ReversedTail).
```

If the clause ordering of append/3 is such that the recursive clause comes before the base clause, then reverse(+, -) will not return with a solution at all. Consider the typical inverse naive program for reverse(+, -) which is of order $O(n^2)$:

```
reverse([], []).
reverse(List, [LastElement|ReversedAllButlast]) :-
    append(AllButLast, [LastElement], List),
    reverse(AllButLast, ReversedAllButLast).
```

Reversing the subgoal ordering in the recursive clause will result in an *inefficient* reverse(+, -) which is of order $O(n^3)$:

```
reverse([], []).
reverse(List, [LastElement|ReversedAllButlast]) :-
    reverse(AllButLast, ReversedAllButLast),
    append(AllButLast, [LastElement], List).
```

The above classification of bugs at different levels of abstraction is similar to the description of programming errors at the code level of [Brna 86b]. There may be a further or alternative description of bugs at the program code error level because of the syntactic closeness of different manifestations of these bugs and there can be many ways of describing and correcting just each bug. It seems that no one classification of bugs can attempt to model the various different and possibly unprincipled ways students (or even Prolog programmers) think about bugs and debug them.

3.2.4 Top-Down Analysis to provide Context

These three levels of bugs relate to the debugging approach of APROPOS2. Given a student's program, APROPOS2 decides :

- which algorithm it knows of for doing the task has been used,
- how the student has decomposed the algorithm into each predicate definition and
- how each predicate definition is implemented for which APROPOS2's analysis can be applied recursively.

APROPOS2 attempts to localise the above kinds of bugs by a process of heuristic code-matching and code-critique, and then to provide a description of the bugs. The bug

description is more than a listing of where the student's incorrect Prolog program differs syntactically from some correct program. As code-matching is done top-down from task statement to algorithm to predicate definition decomposition and then to implementation of each predicate definition, APROPOS2 postulates what the student is intending to do for each line of code he has written. This is in the same sense that PROUST postulates the student's intentions for Pascal programs by code-matching top-down from task goals to plans and then to implementations of these plans. In this way, we have attempted to relate the commentaries to the program context which helps to relate the bugs to possible misconceptions. By representing buggy and inefficient prototypes of common bugs, it can to some extent by retrieving from template commentaries explain why the bug is a bug, that is, why the student program does not work.

3.3 A Quick Case Study : reverse/2

To give a quick overview of the debugging process, we describe how APROPOS2 goes about its task of analysing a student's reverse/2 program. This will be a simplified description of APROPOS2's analysis. More details of its processing will be presented in the other chapters which address specific aspects of the code analysis process. The task and student's solution are :

Task : LIST-REVERSAL

Write a Prolog program reverse/2 which takes a list as input and reverses the elements of the list. A goal call "reverse(X, Y)" should succeed with Y instantiated to the reverse of the list X.

```
append([], L, L).
append([H], L, [H|L]) .
append([H|T], L, Ans) :-
    append(T, L, X),
    append([H], X, Ans).
reverse([], []).
reverse([H], [H]).
reverse([H|T], L) :-
    reverse(T, M),
    append(M, H, L).
```

Figure 3-2 : A buggy solution to reverse/2

The task-specific knowledge that APROPOS2 is provided for the list-reversal task are :

- the intended mode of use is reverse(+, -)
- both arguments of reverse must be of type list
- representations of implementations of algorithms for list-reversal.

APROPOS2 first performs multiple analyses on the program – mode inference, dataflow analysis, type inference and checking, misspelling checking and program transformations. It infers

- an invocation mode of `append(+, +, -)` for the `append` subgoal invocation in the `reverse/2` definition,
- from the predicate definition of `append/3`, that `append/3` needs all its arguments to be of type `list`.

APROPOS2 provides the following program transformation of a student clause to a canonical one to facilitate code-matching at a later stage :

Your clause :
`append([H|T], L, Ans) :-`
 `append(T, L, X),`
 `append([H], X, Ans).`
can be rewritten as :
`append([H|T], L, [H|X]) :-`
 `append(T, L, X).`

APROPOS2 has representations of the three algorithms for list-reversal. Appendix I.1 describes the various algorithms for `reverse/2`. Figure 3-3 shows the key features of the representation of the algorithm for naive reverse.

TASK NAME : reverse/2
ALGORITHM NAME : naive-reverse
LIKELY PREDICATE NAMES : reverse, rev, rv
INVOCATION TYPE OF PREDICATE : both arguments are lists
INVOCATION MODE OF PREDICATE : reverse(+, -)
RECURSION ARGUMENT : list iteration in the 1st argument
NUMBER OF CLAUSES : 2
CLAUSE 1 :
 TYPE : base
 HEAD GOAL : reverse([], [])
 PREFIX SUBGOALS : nil
 RECURSIVE SUBGOALS : nil
 SUFFIX SUBGOALS : nil
 COMMENTARY : "This base case says that the reverse of the empty list is the empty list."
CLAUSE 2 :
 TYPE : recursive
 HEAD GOAL : reverse([H|T], Res)
 PREFIX SUBGOALS : nil
 RECURSIVE SUBGOALS : reverse(T, Sofar)
 SUFFIX SUBGOALS : append(Sofar, [H], Res)
 COMMENTARY : "This recursive case says that the reverse of a non-empty list can be found by reversing the tail of the list and then appending a list consisting only of the first element of the original to the end of the reverse of the tail."
BUGGY-CLAUSE 2 :
 TYPE : recursive
 HEAD GOAL : reverse([H|T], Res)
 PREFIX SUBGOALS : nil
 RECURSIVE SUBGOALS : reverse(T, Sofar)
 SUFFIX SUBGOALS : append(Sofar, H, T)
 COMMENTARY : "This recursive case says that the reverse of a non-empty list can be found by reversing the tail of the list and then appending a list consisting only of the first element of the original to the end of the reverse of the tail. For the append goal to work, it needs arguments which are of type list. In this instance, H is an element which is to be tagged to the end of another list 'Sofar, '. So we need to write append(Sofar, [H], Res)."
... more clause representations ...
CLAUSE ORDERING : no constraint
TEST-CASES :
 [reverse([], Res), Res = []]
 [reverse([a,b], Res), Res = [b,a]]
 [reverse([a,b,c,d,e], Res), Res = [e,d,c,b,a]]
 [append([], [a], Res), Res = [a]]
 [append([b], [a], Res), Res = [b,a]]
 [append([e,d,c,b], [a], Res), Res = [e,d,c,b,a]]

Figure 3-3 : Task information for naive reverse/2

Having done the first part of the analysis, APROPOS2 will then make use of its task knowledge to first recognise the algorithm used in the program. By doing a best-first search of mappings between the student's program and the various algorithm implementations of reverse/2, APROPOS2 selects the naive reverse algorithm as being the most similar to the student's solution. The scoring function computes scores from the closeness of the syntactic match of the programs and features about the programs like modes, types, number of clauses and clause orderings. Code matching is made flexible by matching different permutations of arguments for a certain predicate definition. The code-matching process will be more fully described in Chapter 6.

The student's predicate definition of reverse/2 and append/3 are mapped onto the reference representation of naive reverse/2 and append/3. There is a mismatch between the append(M, H, L) subgoal in the student's program and the reference append(M, [H], L). APROPOS2 will then try to show their equivalence by running them on test cases and comparing the results of their execution. Ideally, what we want is to prove that the set of instantiations at the program point after the invocation of append(M, H, L) in the student's program is equivalent to the set of instantiations at the program point after the invocation of append(M, [H], L) in the reference program. However, this is a difficult problem which is engaging research in the logic programming community [see, for example, Debray 86; Drabent & Maluszynski 86]. In the meantime, we fall back onto a program testing approach which is not foolproof but is at least computationally feasible. We use Shapiro's routines to show equivalences of predicates. Whereas Shapiro's routines have usually been applied to the top-level goal invocation, we use heuristic code-matching to suggest parts of the code that actually differ, localise these lower-level predicates and show their equivalence with the corresponding parts of the reference program, and if non-equivalent, Shapiro's diagnosis routines can be used to further localise the source of the error.

The mode generation module of APROPOS2 inferred an invocation mode of append(+, +, -) for the student program. The type inference module of APROPOS2 inferred that M is of type list and H may not be of type list, that is, may be just an atom. The dynamic analysis module runs the student predicate append/3 on appropriate test data from the task information. Mode and type information is used to check the validity of the instantiations in the student predicate invocation – append(M, H, L), that is, that M is to be instantiated to a list, H can be instantiated to an atom and L is a result variable. In our earlier version, APROPOS1 where no test data is stored with the task information, the type information inferred is used to select suitable data to instantiate '+' arguments. Since L is the only result variable, the dynamic analysis module checks if running the student predicate will instantiate L to the same value as running the reference predicate does.

In this case, the dynamic analysis module fails to find equivalence, so it concludes a bug of an incorrect subgoal has been made. Note that in running `append(M, H, T)`, the student's predicate definition is used, and in running `append(M, [H], T)`, the reference predicate definition is used. If, for example, the student's solution for `append/3` had been

```
append([], L, [L]).
append([H|T], L, [H|X]) :-
    append(T, L, X).
```

then the student's subgoal of `append(M, H, L)` would have been accepted as correct, even though there has been no explicit representation of this student's `append/3` program in APROPOS2. If, as well, the student has instead named his `append/3` as `attach/3` or even `attach/4`, then APROPOS2 will remember a heuristic match between the student's `attach/3` or `attach/4` and the reference's `append/3` so that `append/3` can then be used to answer oracle queries about `attach/3` or `attach/4`. If the student's predicate definition for `append/3` is identical to the reference's predicate definition of `append/3`, then APROPOS2 will conclude non-equivalence between `append(M, H, L)` and `append(M, [H], L)`.

Here then is the result of the analysis :

Your program seems to be incorrect.
Now, this is APROPOS2's critique on your program :

Clause 1 of `reverse/2` seems fine.
This base case says that the reverse of the empty list is the empty list.

Clause 2 of `reverse/2`
 `reverse([H], [H]).`
seems redundant.

In Clause 3 of `reverse/2`,
 `append(M, H, L)` is incorrect.
Replace it by `append(M, [H], L)`.
This recursive case says that the reverse of a non-empty list can be found by reversing the tail of the list and then appending a list consisting only of the first element of the original to the end of the reverse of the tail. For the `append` goal to work, it needs arguments which are of type list. In this instance, `H` is an element which is to be tagged to the end of another list `M`. So we need to write `append(M, [H], T)`.

Clause 1 of `append/3` seems fine.
This base case says that the result of appending an empty list to a list `L` is the list `L`.

Clause 2 of `append/3`
 `append([H], L, [H|L]).`
seems redundant.

Clause 3 of `append/3` seems fine.
This recursive case says that the result of appending a list `[H|T]` to a list `X` is obtained by appending `[H]` to the result of appending the list `T` to `X`.

Figure 3-4 : APROPOS2's analysis of a student's naive `reverse/2` program

It is possible to detect `append(H, L, [H|L])` and `reverse([H], [H])` are redundant at the program transformation stage. This would require a more complete program transformation system which currently does not exist in APROPOS2.

The above analysis tells what is wrong with the student code. To explain why it is wrong, APROPOS2 does a further dynamic analysis using the diagnosis routines of Shapiro's and matched reference predicates serving as oracles. We have :

On running your program on an example : `reverse([a,b], X)`.
It produces the solution : `reverse([a,b], [b|a])` which is incorrect.

The source of the error is :

```
reverse([a,b], [b|a]) :-  
    reverse([b], [b]),  
    append([b], a, [b|a]).
```

The goals `reverse([b], [b])` and `append([b], a, [b|a])` are correct as we want them to be, but `reverse([a,b], [b|a])` is not true. What we want is `reverse([a,b], [b,a])`.

Do you want to trace through the execution of this?

(If student types yes, APROPOS will lead student using a screen-oriented tracer which steps through the control flow and shows each instantiation of a predicate argument as it occurs. The tracer SODA [Plummer 85] is adapted for this purpose)

Figure 3-5 : APROPOS2's further analysis of student's naive reverse/2 program

3.4 Summary

We have discussed the use of multiple sources of program analyses in analysing Prolog programs in APROPOS2. Information garnered from these sources of expertise is used in the heuristic code-matching process in APROPOS2 which will be detailed in Chapter 5. We have described the different levels of bugs APROPOS2 detects and discussed the debugging process of APROPOS2 in its analysis of an actual student's program. This case study shows how we have integrated code analysis with dynamic analysis assisted by using mode and type information derived from APROPOS2's different sources of program analyses. It also shows how an algorithm-based analysis of Prolog predicate definitions provides the context for description of the function of each definition and each of its component clauses.

Chapter 4. A Frame Representation for Prolog Predicate Definitions

In this chapter, we discuss the representations that APROPOS2 uses for programs and implementations of algorithms. The factors which affect the choice of representations are :

- they need to allow algorithms to be matched against the representations of students' programs,
- they need to allow for alternative implementations of an algorithm to be recognised as far as possible, and features of programs to be compared and criticised,
- they need to allow meaningful contextual English-language descriptions of the algorithms used, their decompositions into code and the bugs involved, to be generated,
- they need to be parsimonious.

4.1 P-frame Representation

This section discusses the representations that APROPOS2 uses for programs and implementations of algorithms. APROPOS2 uses frames to represent both student predicate definitions and reference predicate definitions. Implementations of an algorithm for a predicate definition are represented as a Prolog-frame (or **P-frame** in short). APROPOS2 does not directly match the student's Prolog program. Instead, it converts each student predicate definition to a P-frame and matches these student P-frames against a library of prestored reference P-frames for the problem task.

4.1.1 Reference P-frames

A reference P-frame contains abstract computational features of a Prolog program. It has slots for :

- task name
- algorithm name
- possible likely names for predicate
- type information about predicate
- mode information about predicate
- number of arguments in predicate
- number of clauses used
- recursion type : list, tree, number recursion or unknown; single or multiple recursion

- programming techniques used
- canonical representation for each clause or its alternatives, and common buggy versions
- for each correct clause, a commentary template to describe its function and for each incorrect clause, why it is incorrect
- clause ordering constraint
- test cases for running subgoal predicates with expected results

For each task, there can be several algorithms. Each algorithm is implemented in a predicate definition which is represented as templates for clauses in which predicate and variable names are replaced by second-order variables. The representations of predicate definitions as its alternative clause decompositions with their associated commentary templates provide models of the functions underlying each predicate definition and each clause in the definition.

To illustrate the representation used, consider the reverse/2 task again. There are 3 reference P-frames, one for each correct algorithm. Figures 4-1 and 4-2 below show the P-frames for naive and inverse naive reverse. Since both these algorithms require the use of the append/3, we also require P-frame(s) for the append/3 task. We will however not show the append/3 P-frame here.

P-FRAME :

TASK NAME : reverse/2

ALGORITHM NAME : naive-reverse

LIKELY PREDICATE NAMES : reverse, rev, rv

NUMBER OF ARGUMENTS : 2

INVOCATION TYPE OF PREDICATE : both arguments are lists

INVOCATION MODE OF PREDICATE : reverse(+, -)

RECURSION ARGUMENT : list iteration in the 1st argument

PROGRAMMING TECHNIQUES : naive-recursion

NUMBER OF CLAUSES : 2

CLAUSE 1 :

TYPE : base

HEAD GOAL : reverse([], [])

PREFIX SUBGOALS : nil

RECURSIVE SUBGOALS : nil

SUFFIX SUBGOALS : nil

COMMENTARY : "This base case says that the reverse of the empty list is the empty list."

CLAUSE 2 :

TYPE : recursive

HEAD GOAL : reverse([H|T], Res)

PREFIX SUBGOALS : nil

RECURSIVE SUBGOALS : reverse(T, Sofar)

SUFFIX SUBGOALS : append(Sofar, [H], Res)

COMMENTARY : "This recursive case says that the reverse of a non-empty list can be found by reversing the tail of the list and then appending a list consisting only of the first element of the original to the end of the reverse of the tail."

CLAUSE 2 :

TYPE : recursive

HEAD GOAL : reverse([H|T], Res)

PREFIX SUBGOALS : append(Sofar, [H], Res)

RECURSIVE SUBGOALS : reverse(T, Sofar)

SUFFIX SUBGOALS : nil

COMMENTARY : 'This recursive case says that the reverse of a non-empty list can be found by reversing the tail of the list and appending a list consisting only of the first element of the original to the end of the reverse of the tail. However, you should put the ', append/3, ' subgoal after the recursive subgoal for efficiency reasons. If it comes before the recursive subgoal, then ', Sofar, ' and ', Res, ' will be uninstantiated variables when this goal is attempted. It will try many values for ', Sofar, ' and ', Res, ' before it gets the right one.'

BUGGY-CLAUSE 2 :

TYPE : recursive

HEAD GOAL : reverse([H|T], Res)

PREFIX SUBGOALS : nil

RECURSIVE SUBGOALS : reverse(T, Sofar)

SUFFIX SUBGOALS : append(Sofar, H, Res)

COMMENTARY : 'This recursive case says that the reverse of a non-empty list can be found by reversing the tail of the list and then appending a list consisting only of the first element of the original to the end of the reverse of the tail. For the append goal to work, it needs arguments which are of type list. In this instance, H is an element which is to be tagged to the end of another list ', Sofar, '. So we need to write append(Sofar, [H], Res).'

CLAUSE ORDERING : no constraint

TEST-CASES :

[reverse([], Res), Res = []]

[reverse([a,b], Res), Res = [b,a]]

[reverse([a,b,c,d,e], Res), Res = [e,d,c,b,a]]

[append([], [a], Res), Res = [a]]

[append([b], [a], Res), Res = [b,a]]

[append([e,d,c,b], [a], Res), Res = [e,d,c,b,a]]

Figure 4-1 : Reference P-frame for naive algorithm for doing reverse/2

P-FRAME :
TASK NAME : reverse/2
ALGORITHM NAME : inverse-naive-reverse
LIKELY PREDICATE NAMES : reverse, rev, rv
NUMBER OF ARGUMENTS : 2
INVOCATION TYPE OF PREDICATE : both arguments are lists
INVOCATION MODE OF PREDICATE : reverse(+, -)
RECURSION ARGUMENT : list iteration in the 2nd argument
PROGRAMMING TECHNIQUES : tail-recursion
NUMBER OF CLAUSES : 2
CLAUSE 1 :
 TYPE : base
 HEAD GOAL : reverse([], [])
 PREFIX SUBGOALS : nil
 RECURSIVE SUBGOALS : nil
 SUFFIX SUBGOALS : nil
 COMMENTARY : "This base case says that the reverse of the empty list is the empty list."
CLAUSE 2 :
 TYPE : recursive
 HEAD GOAL : reverse(List, [Last|Sofar])
 PREFIX SUBGOALS : append(Restlist, [Last], List)
 RECURSIVE SUBGOALS : reverse(Restlist, Sofar)
 SUFFIX SUBGOALS : nil
 COMMENTARY : "This recursive case says that the reverse of the non-empty list can be found by reversing the list without the last element, and then appending a list containing only of the last element of the original list with the result or reversing the list without the last element."
CLAUSE ORDERING : no constraint
TEST-CASES :
 [reverse([], Res), Res=[]]
 [reverse([a,b], Res), Res=[b,a]]
 [reverse([a,b,c,d,e], Res), Res=[e,d,c,b,a]]
 [append(X, [Y], [a]), X=[], Y=a]
 [append(X, [Y], [a,b]), X=[a], Y=b]
 [append([X, [Y], [e,d,c,b,a]), X=[e,d,c,b], Y=a]

Figure 4-2 : Reference P-frame for inverse naive algorithm for doing reverse/2

A P-frame represents an equivalence class of implementations of an algorithm. With a P-frame, we can generate the various possible implementations (both correct and incorrect) of the algorithm represented by the P-frame by selecting for each i between 1 and **NUMBER OF CLAUSES**, one of the different **CLAUSES** i or **BUGGY-CLAUSES** i and satisfying any clause ordering constraints. In fact, the class of possible implementations represented by the P-frame is more than this, since we also include implementations whose computational equivalence between them and a canonical representation

- can be shown by dynamic analysis, or
- can be established using properties of Prolog primitives, like the commutative and associative properties of arithmetic operators or by a database of equivalent Prolog code, or
- can be shown by applying folding and unfolding program transformation rules.

4.1.2 Student P-frames

A student's Prolog program is first parsed to convert all variable names to special atomic names to avoid unintended instantiation of variables and then transformed to an internal representation. APROPOS2 does the first part of its analysis (dataflow, mode and type inference, etc) on this internal representation. Features are abstracted from this analysis to construct a student P-frame, the structure of which is identical to the reference P-frame except it obviously does not have some slots like algorithm name which is unknown yet and test cases. To illustrate, APROPOS2 parses this student's program for `reverse/2` (and `getlastrest/3`) written for `reverse(+, -)` into its internal representation and constructs these 2 P-frames :

```

reverse([], []).
reverse(L, [Last_El|Rev_Els]) :-
    getlastrest(L, Last_El, Other_Els),
    reverse(Other_Els, Rev_Els).

getlastrest([Last_El], Last_El, []).
getlastrest([A|Rest], Last_El, [A|Other_Els]) :-
    getlastrest(Rest, Last_El, Other_Els).

```

P-FRAME :
TASK NAME : reverse/2
PREDICATE NAME : reverse
NUMBER OF ARGUMENTS : 2
INVOCATION TYPE OF PREDICATE : both arguments are lists
INVOCATION MODE OF PREDICATE : reverse(+, -)
RECURSION ARGUMENT : list iteration in the 2nd argument
PROGRAMMING TECHNIQUES DETECTED : tail-recursion
NUMBER OF CLAUSES : 2
CLAUSE 1 :
 TYPE : base
 HEAD GOAL : reverse([], [])
 PREFIX SUBGOALS : nil
 RECURSIVE SUBGOALS : nil
 SUFFIX SUBGOALS : nil
CLAUSE 2 :
 TYPE : recursive
 HEAD GOAL : reverse(L, [Last_El|Rev_Els])
 PREFIX SUBGOALS : getlastrest(L, Last_El, Other_Els)
 RECURSIVE SUBGOALS : reverse(Other_Els, Rev_Els)
 SUFFIX SUBGOALS : nil

P-FRAME :
TASK NAME : getlastrest/3
PREDICATE NAME : getlastrest
NUMBER OF ARGUMENTS : 3
INVOCATION TYPE OF PREDICATE : getlastrest(list(atom), atom, list(atom)) (inferred by APROPOS2)
INVOCATION MODE OF PREDICATE : getlastrest(+, -, -) (inferred by APROPOS2)
RECURSION ARGUMENT : list iteration in the 1st and 3rd argument
PROGRAMMING TECHNIQUES DETECTED : tail-recursion
NUMBER OF CLAUSES : 2
CLAUSE 1 :
 TYPE : base
 HEAD GOAL : getlastrest([Last_El], Last_El, [])
 PREFIX SUBGOALS : nil
 RECURSIVE SUBGOALS : nil
 SUFFIX SUBGOALS : nil
CLAUSE 2 :
 TYPE : recursive
 HEAD GOAL : getlastrest([A|Rest], Last_El, [A|Other_Els])
 PREFIX SUBGOALS : nil
 RECURSIVE SUBGOALS : getlastrest(Rest, Last_El, Other_Els)
 SUFFIX SUBGOALS : nil

Figure 4-3 : Student's program and P-frames for inverse naive reverse/2

If program transformation rules have been applied to a student clause to fold away subgoal(s), APROPOS2 will have representations of both clauses in the student P-frame. For example, for the student clause :

```
append([H|T], L, Ans) :-
    append(T, L, X),
    append([H], X, Ans).
```

there will be these two slots in the student P-frame for append/3 :

CLAUSE 3 :

```
TYPE : recursive
HEAD GOAL : append([H|T], L, Ans)
PREFIX SUBGOALS : nil
RECURSIVE SUBGOALS : append(T, L, X), append([H], X, Ans)
SUFFIX SUBGOALS : nil
```

FOLDED-CLAUSE 3 :

```
TYPE : recursive
HEAD GOAL : append([H|T], L, [H|X])
PREFIX SUBGOALS : nil
RECURSIVE SUBGOALS : append(T, L, X)
SUFFIX SUBGOALS : nil
```

This allows for more flexible matching. For instance, it may be the case that because of code errors in the student's program, a folding operation is made possible, and throwing away the unfolded version will adversely affect the matching. With representations of the folded version and the original unfolded version, APROPOS2 finds which of them when matched to a reference clause has the better fit.

4.1.3 Matching Student P-frames to Reference P-frames

If a student P-frame is best-matched to a reference P-frame, then APROPOS2 infers that it has uncovered that the intended function of the student predicate definition as represented in the student P-frame is that of the reference predicate definition as represented in the reference P-frame. Similarly, within this matching of P-frames, if a clause in the student P-frame is best-matched to a clause in the reference P-frame, then APROPOS2 infers that it has uncovered that the intended function of the student clause is that of the matched reference clause in the reference P-frame.

4.2 Algorithms for a Task

For each programming task, there can be several algorithms for doing the task. An algorithm is implemented as a predicate definition, and may invoke new tasks. For example, there are the bubble sort, insertion sort, permutation sort and quick sort algorithms for sorting a list.

The permutation sort invokes the new task of permutation of a list which itself can be implemented using different algorithms.

Consider the 3 algorithms for doing reverse/2 again : the *naive* method, the *railway-shunt* method, and the *inverse naive* method. The naive method is really the inverse naive method if we swap the two arguments of the reverse goals. We choose to distinguish between them as the task posed to students is to write reverse(+,?). We have included the method of reverse/2 using difference lists in the railway-shunt method as their dataflow are similar.

In computer science terminology, an *algorithm* is defined as a method of solving a problem which must be studied, not the computer implementation, to learn how the problem is being solved [Sedgewick 83]. If we are to represent algorithms based on this definition, we have to look into the process of converting the algorithm in some language-independent notation into Prolog code. This is another research issue in itself. Thus we have not represented algorithms in this way. We have only implementations of algorithms in the reference P-frames. However, we make use of this definition of algorithm to characterise our reference P-frames. For each algorithm we think of for a task, we will create a P-frame which contains all the possible correct and incorrect ways we can think for implementing the algorithm.

Consider the replace_elements/4 task. The problem statement is :

Task : replace_elements/4

Write a Prolog program replace_elements/4 which replaces all occurrences of a given element in a list by another and instantiates a given variable to the answer. The arguments should be, in order :

- 1) the element to be replaced
- 2) what to replace it with
- 3) the given list
- 4) a variable to be instantiated to the answer.

Your predicate should not bother to delve inside lists within lists.

How many algorithms are there for this task? This depends on how abstract one sees a strategy for implementing replace_elements/4. In the most abstract, there is only one : check if the head of the input list is identical to the element to be replaced and if so, replace it by the element to be replaced with and recursively work on the tail of the list. If we adopt this abstract representation for program analysis, it will be difficult to generate useful advice for a specific implementation. The task for an automated program analyser is to recognise the different implementations of an algorithm and provide useful pertinent advice concerning a specific implementation. For this reason, we choose to identify 3 algorithms for the task to be represented as P-frames. These 3 algorithms are *appropriate to the Prolog execution model*. We use the word *algorithm* for lack of a better word, and our meaning of the word is different from that of Sedgewick.

The first algorithm is the *procedural* version which checks if the head of the input list is identical to the element to be replaced and if so, replaces it by the element to be replaced with and recursively works on the tail of the list. A typical program for this is :

```
replace_elements(_, _, [], []).
replace_elements(Old, New, [Old|T], [New|Sofar]) :-
    !,
    replace_elements(Old, New, T, Sofar).
replace_elements(Old, New, [H|T], [H|Sofar]) :-
    replace_elements(Old, New, T, Sofar).
```

There is a clause ordering constraint in the representation of this algorithm, which is that the second clause must come before the third clause. We label this program procedural as it works because of its procedural meaning. Removing the cut will, for example, incorrectly produce two solutions for a call `replace_elements(a, b, [a], Res)`, namely, `Res=[b]` and `Res=[a]`.

The second algorithm is the *declarative* version of the first or second algorithm. A typical program for this is :

```
replace_elements(_, _, [], []).
replace_elements(Old, New, [Old|T], [New|Sofar]) :-
    replace_elements(Old, New, T, Sofar).
replace_elements(Old, New, [H|T], [H|Sofar]) :-
    Old \= H,
    replace_elements(Old, New, T, Sofar).
```

The third algorithm is the *unfolded* version which calls an auxiliary predicate to the job of replacing each element. A typical program for this is :

```
replace_elements(_, _, [], []).
replace_elements(Old, New, [Inp|T], [Out|Sofar]) :-
    replace_one(Old, New, Inp, Out),
    replace_elements(Old, New, T, Sofar).
```

There are two algorithms for `replace_one/4`, namely, the procedural version :

```
replace_one(Old, New, Old, New) :-
    !.
replace_one(Old, New, H, H).
```

and the declarative version :

```
replace_one(Old, New, Old, New).
replace_one(Old, New, H, H) :-
    Old \= H.
```

What criteria do we use to single out a class of programs as an algorithm to be represented in a P-frame? The programs in the class should use one common strategy of solving the task. The strategies have to take into account the procedural semantics of Prolog. All of the programs in a class can be explained in a common way. This is important as it allows us to write specific template commentaries to explain the strategy, explain the function of each correct clause, and explain why a buggy clause is incorrect. To illustrate, the template commentary of the recursive clause of the naive reverse algorithm is :

"This recursive case says that the reverse of a non-empty list can be found by reversing the tail of the list and then appending a list consisting only of the first element of the original to the end of the reverse of the tail."

and the template commentary of a buggy recursive clause of the naive reverse algorithm is :

"This recursive case says that the reverse of a non-empty list can be found by reversing the tail of the list and then appending a list consisting only of the first element of the original to the end of the reverse of the tail. For the append goal to work, it needs arguments which are of types list. In this instance, H is an element which is to be tagged to the end of another list Sofar. So we need to write `append(Sofar, [H], Res)`."

We also characterise programs as being of one algorithm rather than another in order to facilitate the code-matching process. While the third algorithm of `replace_elements/4` can really be folded into the first or second algorithm, we single it out as another algorithm so as not to convolute the code-matching process. Another approach is to have a canonical representation of a more general algorithm and to apply program transforms to this general algorithm to synthesize the various possible variants and implementations of the algorithm. The problems with this approach are :

- the difficulty of explaining each variant in terms of the context of the general algorithm, and
- the difficulty of writing program transforms that apply to a wide class of Prolog programs.

These issues also suggest the kind of trade-offs posed when we decide to choose more abstract canonical representations of programs as against less abstract and more specific ones. In this sense, we have compiled the different variants of the one general algorithm for `replace_elements/4` as different algorithms for a better explanation facility for tutoring purposes, and for efficiency's sake. Another consideration for the way we characterise our algorithms is that we want to have APROPOS2 do as much code-matching as possible instead of reasoning about code which will be necessary if we use more parsimonious abstract representations.

4.3 How Knowledge in P-frames is Derived

We have described how we have codified knowledge about algorithms in APROPOS2. We now discuss how, for a certain programming task, we have derived the programming and task knowledge to codify implementations of the algorithms. The basis for our knowledge representations is :

- (1) introspection,
- (2) studies of programs written for the tasks gathered from Prolog code libraries and Prolog textbooks, and
- (3) studies of available collected programs students wrote for the task.

We have encoded reference P-frames in APROPOS2 for 5 tasks. They are :

- reverse/2 to reverse a list
- sorting/2 to sort a list
- replace_elements/4 to replace all occurrences of one element in a list by another
- atom_count/2 to count the number of atoms in a list
- fringe_length/2 to count the number of atoms in all levels of a list

The sources of derivation of the P-frames for reverse/2 are : (1), (2) and studies of an initial set of 40 hand-written solutions to the task. The sources of derivation of the P-frames for sorting/2 are : (1) and (2). The sources of derivation of the P-frames for the other tasks are : mostly (1) to think of all the possibilities a program for the task can be written, (2) and reviewing the first 5 to 10 collected solutions for each task. Having created these P-frames, we need to test APROPOS2 on these initial programs as well as new ones. Such empirical evaluation will help us find out whether APROPOS2's representations of algorithms can be realistically used to match students' programs to recognise the algorithms used in them and debug them. Our tests of APROPOS2, described in Chapter 7, give us reason to believe that APROPOS2's knowledge representations do serve the purpose of providing useful analyses of student's Prolog programs.

In summary, we have described the representations of programs in APROPOS2. In the next chapter, we will describe how these representations are used in the code-analysis approach of APROPOS2.

Chapter 5. Algorithm Recognition, Bug Detection and Bug Correction

There are 2 phases in the task-specific analysis of APROPOS2. The first phase is the *code-matching* phase in which APROPOS2 does a best-first search to find the best mappings of student predicate definitions (*SPDs*) to reference predicate definitions (*RPDs*). The second phase is the *code-critique* phase in which APROPOS2 tries to account for any discrepancy between the student code and the correct code. In each pairing of predicate definitions, APROPOS2 tries to establish equivalence of function by first trying code-matching and then dynamic analysis. Non-equivalence indicates the presence of bugs. Any differences that cannot be explained away will be treated as bugs. Corrections to these bugs will be suggested from the code mappings derived from the first phase.

For brevity in description, we will use the terms *student predicates*, *student subgoals* and *student terms* to refer to the predicate names, subgoals and terms that appear in the student's program, and *reference predicates*, *subgoals* and *terms*, to refer to the predicate names, subgoals and terms that appear in the reference program, dispensing with the use of the possessive 's.

5.1 The Search Space

This section describes how APROPOS2 recognises algorithms in a student program and matches the student P-frame to a reference P-frame. It pairs student predicate definitions with reference predicate definitions. These mappings are important as they allow APROPOS2 to detect and correct bugs by static and dynamic analyses. APROPOS2 performs algorithm recognition by searching the best reference P-frames to match the student P-frames. It uses a heuristic scoring function to compute a weighted sum of P-frame slot differences to evaluate how well two P-frames match. Each state of the search will be a partial mapping – some SPDs have been matched to RPDs while other SPDs remains to be matched at later stages of the search. The partial matching relies on both syntactic and semantic information. The search ends when all the SPDs have been matched – a complete mapping.

Each algorithm for a task is represented by a set of reference P-frames each of which in turn associates with new tasks. Each of these new tasks has a set of algorithms to implement them. The complete search space is all the possible mappings from the student P-frames to the library of reference P-frames. To constrain the search space, we use heuristics to guide the search.

Each state in the search space represents a series of mappings from student predicate definitions to reference ones. Each reference predicate definition is always associated with one of the stored algorithms. Suppose the student program comprises n predicate definitions, SPD1, SPD2, ..., SPD n , of which SPD1 is the top level one, and suppose the problem task has three stored algorithms A, B and C. APROPOS2 will match the top level SPD1 first with each of the reference P-frames for A, B, and C, or to NOMATCH (which means no good match for SPD1). Each partial matching of SPD1 with A, B or C will suggest which tasks might match SPD2, ..., SPD n . APROPOS2 selects the reference P-frame which has the best fit according to the scoring function to be described further. Suppose B is chosen. Then the next match to do is selecting one of SPD2, ..., SPD n and matching it with all the algorithms of the task as suggested by the match of SPD1 with B. If APROPOS2 matches the top level SPD1 with NOMATCH, the problem arises as to how to proceed with the search. In such a case, APROPOS2 will match the next student SPD with the reference P-frames for A, B and C instead.

Each state is recorded as a node. A node is closed when all its successors have been created as successor nodes, otherwise it is an open node. At each step in the search, the best node is the open node with the lowest score and is expanded. Each node suggests what task each unmapped student predicate definition may be mapped to. A node is expanded by mapping the next unmapped student predicate definition to all possible algorithms of its matched task. This can generate n nodes if there are n possible algorithms for the matched task. Each match of a student predicate definition with a reference predicate definition may constrain the possible matches of the as-yet-unmapped student predicate definitions by mapping them to reference tasks.

APROPOS2 also generates mappings of student predicate definitions and reference tasks to NOMATCH. In the first case there is no match for a student predicate definition and so it is considered redundant to a correct solution. In the second case there is no match for a reference task which is considered essential to a correct solution but missing from the student's solution. The search terminates when there is an open node with the best score which cannot be expanded, that is, the mapping is complete. This will then be the winning node.

To illustrate, in matching the student's sorting/2 program in Figure 5-4, APROPOS2 first matches the sorting/2 SPD to all the RPDs for the sorting/2 task and NOMATCH as shown in Figure 5-6. It selects the best match which is the insert-sort RPD which suggests that the insert/3 SPD should be matched to the RPDs for insert/3 reference task and also to NOMATCH. After this matching, the mapping is complete as *all* the SPDs have been matched to some RPDs or to NOMATCH.

Figure 5-1 shows a schematic diagram of the algorithm matching process in APROPOS2.

5.2 Best-First Search

The representation of a node in APROPOS2 is a frame with the following slots :

- **NODE NUMBER** : the chronological order of the node creation
- **NODE SCORE** : score of best complete mapping that can result from the partial mapping done in this node
- **SPD/RPD MAPPINGS** : a list of mappings of SPDs with RPDs with the following information for each mapped SPD :
 - the mapped task
 - the mapped algorithm RPD
 - the best permutation argument order of the RPD
 - mappings of SPD clauses to RPD clauses
- **UNMAPPED SPDs** : a list of as yet unmapped SPDs
- **SUGGESTED UNMAPPED-SPD/TASK MAPPINGS** : an ordered list of suggested pairings of as-yet-unmatched SPDs to reference tasks.

We first briefly explain the function of each slot. The **NODE NUMBER** is used for documentation purposes. The **NODE SCOREs** of open nodes are used to decide which node is to be next expanded. The **SPD/RPD MAPPINGS** of a node is the record of the partial mapping so far achieved in the node. The **SPD/RPD MAPPINGS** of the winning node will provide all the information for the next phase of APROPOS2's analysis - code-critique. The **UNMAPPED SPDs** will be accessed to find out which SPD is to be matched next. Having selected this SPD, the **SUGGESTED UNMAPPED-SPD/TASK MAPPINGS** is used to find out which reference task has been associated with the SPD.

The node is implemented as a term in the internal database using the record/3 predicate in Edinburgh Prolog. APROPOS2 always starts off by creating this initial open node for the problem task TASK :

- **NODE NUMBER** : 0
- **NODE SCORE** : 0
- **SPD/RPD MAPPINGS** : []
- **UNMAPPED SPDs** : a list of all the SPDs
- **SUGGESTED UNMAPPED-SPD/TASK MAPPINGS** : [TASK/TASK]

Figure 5-2 shows the algorithm used in the best-first search.

Match Programs

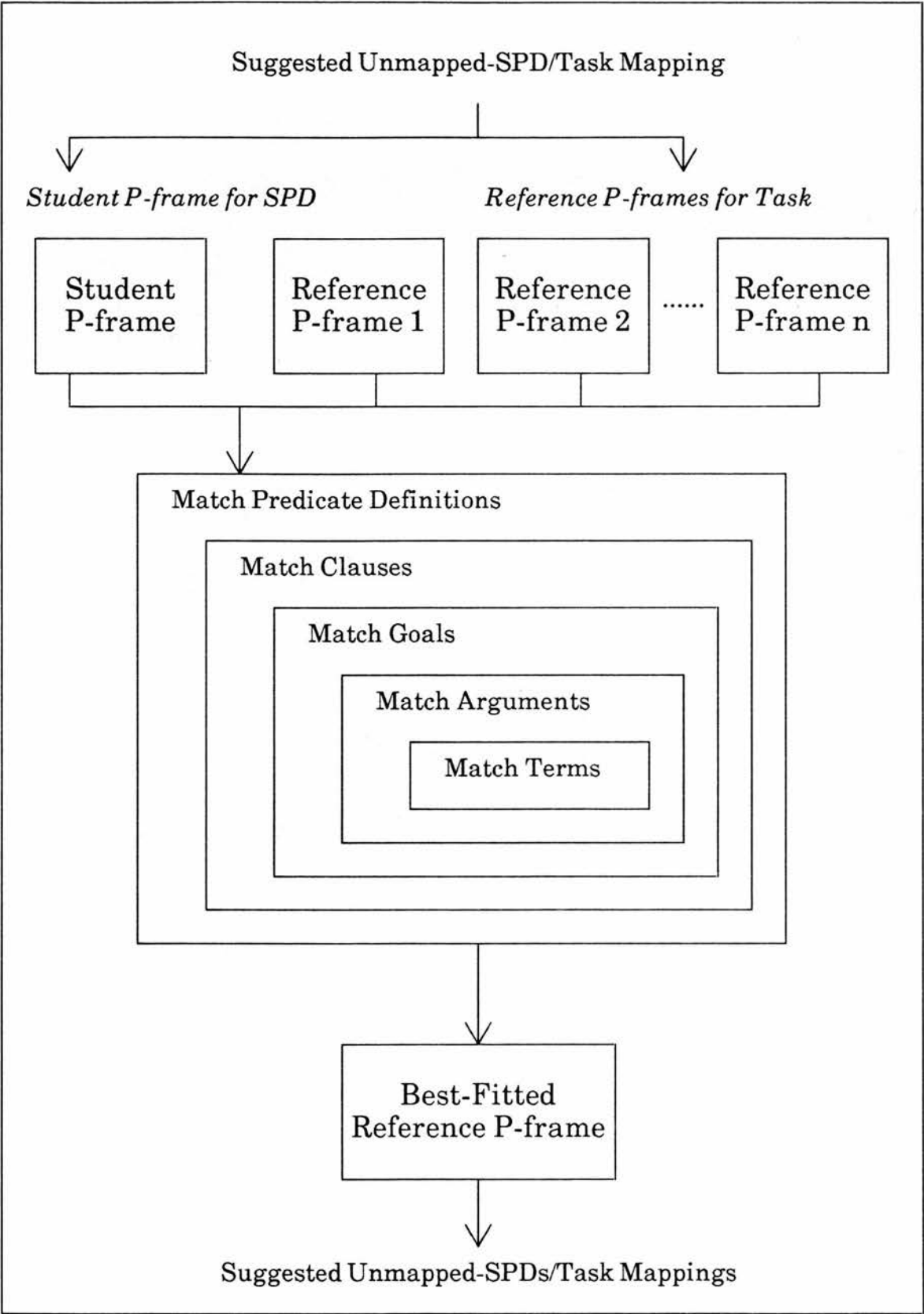


Figure 5-1 : Algorithm recognition in APROPOS2

Step 1 :

Fetch open node with lowest score. Suppose it is

- **NODE NUMBER :** *NODE NUMBER*
- **NODE SCORE :** *NODE SCORE*
- **SPD/RPD MAPPINGS :** *SPD/RPD MAPPINGS*
- **UNMAPPED SPDs :** *UNMAPPED SPDs*
- **SUGGESTED UNMAPPED-SPD/TASK MAPPINGS :** *SUGGESTED UNMAPPED-SPD/TASK MAPPINGS*

Step 2 :

If *SUGGESTED UNMAPPED-SPD/TASK MAPPINGS* is empty, then node is the best interpretation so far and exit,

else from *SUGGESTED UNMAPPED-SPD/TASK MAPPINGS*, select the first *UNMAPPED-SPD/TASK* pair.

Fetch all reference P-frames for *TASK* (the different algorithms for *TASK*).

Step 3 :

For each reference P-frame,

match the student P-frame to the reference P-frame,

collect results of match into *NEW SPD/RPD MAPPING*

compute the score for this partial mapping,

compute the score *NEW NODE SCORE* of best complete mapping that can result from this partial mapping,

create a new open node :

- **NODE NUMBER :** *updated NODE NUMBER*
- **NODE SCORE :** *NEW NODE SCORE*
- **SPD/RPD MAPPINGS :** *SPD/RPD MAPPINGS & NEW SPD/RPD MAPPING*
- **UNMAPPED SPDs :** *UNMAPPED SPDs with SPD deleted*
- **SUGGESTED UNMAPPED-SPD/TASK MAPPINGS :** *NEW SUGGESTED UNMAPPED-SPD/TASK MAPPINGS with the UNMAPPED-SPD/TASK pair deleted.*

Close this node.

Step 4 :

Go to *Step 1*.

Figure 5-2 : Algorithm for APROPOS2's best-first search

5.3 Measuring P-frame Mismatch

Our scoring function uses a penalty score. The lower the score, the better the match is. A score of 0 implies a perfect match of computational equivalence. The best-first search implemented in APROPOS2 is the A* algorithm described in [Hart 68; Hart 72]. The goal state for the search is a complete mapping of SPDs to RPDs. The scoring function used is :

$$f[S] = g[S] + h[S]$$

where

- S is a node in the best-first search. Each node represents a partial mapping of SPDs to RPDs. For the winning node, it will be a complete mapping of SPDs to RPDs.
- f is the cost estimate of the best complete mapping that can result from extending the partial mapping in S to a complete mapping,
- g is the direct cost measure of the PDs mappings in S,
- h is the estimate of the cost of extending the mapping in S to a complete mapping.

The choice of a good searching algorithm like best-first search is critical because each node expansion, mainly the computation of g, is computationally expensive even though the number of node expansions is unlikely to be large for the kind of number and list manipulation tasks APROPOS2 handles. Our computation of h is as follows :

for each SPD in *UNMAPPED SPDs* which has a *SUGGESTED UNMAPPED SPD/TASK MAPPING* (which means that SPD has been mapped to a reference task TASK), add 0.
{ *This is an optimistic estimate that unmapped SPDs will find perfect matches later on in the search. Thus it always never overestimates the true score.* }

for each SPD in *UNMAPPED SPDs* which has a *SUGGESTED UNMAPPED SPD/[] MAPPING* (which means that SPD has been mapped to NOMATCH), add 50.
{ *This is a true estimate that the unmapped SPD will eventually be matched to NOMATCH with a score of 50. Again, it never overestimates the cost.* }

As the function h used by APROPOS2 never overestimates the actual cost, the best-first search is admissible [Nilsson 80] ensuring that the best mapping, in terms of APROPOS2's measure of code mismatch, will always be found. The scoring function g is used to estimate the quality of the predicate definition mapping of a node. It is computed from a sum of **match**(S-P-frame, R-P-frame) which measures how well the student P-frame matches the reference P-frame. This match measure is computed by functions that measure the differences between the P-frames' slots and the differences are weighted according to the semantics of the slots.

We have based the weighting and scoring scheme on some structure by scoring mismatches on a hierarchy of levels. The idea is that mismatches at the predicate definition level penalise

more than mismatches at the clause level, which in turn count more than mismatches at the subgoal level, and then the argument and term levels. The weighting and scoring scheme are arrived at by running APROPOS2 for the reverse/2 task on the initial set of 40 collected hand-written programs and tuning the scores.

We set the threshold for a valid match of a SPD to a RPD to be 50. Thus if a SPD is matched to a RPD with a score of more than 50, then this match will be considered no better than the match of SPD to NOMATCH.

Here are the initial scores, adding up to **match**(S-P-frame, R-P-frame), which measure the matching of general features of predicate definitions :

```
% compare predicate names
If predicate name (S-P-frame) matches one of likely predicate names(R-P-frame)
then add 0 else 4.

% compare no. of arguments in predicate
Let no. of arguments (S-P-frame) be NAS and no. of arguments (R-P-frame) be NAR.
If NAS = NAR then add 0
else if NAS > NAR then add 4*(NAS-NAR)
else add 4*(NAR-NAS).

% compare recursion arguments
If recursion argument (S-P-frame) is identical to recursion argument (R-P-frame)
then add 0 else 5.

% compare no. of clauses in predicate definition
Let no. of clauses (S-P-frame) be NCS and no. of clauses (R-P-frame) be NCR.
If NCS = NCR then add 0
else if NCS > NCR then add 2*(NCS-NCR)
else add 3*(NCR-NCS).
```

At this stage, the SPD will be matched against all the possible permutations of arguments in the RPD. Section 5.4 discusses why we want to do this. For each match of the S-P-frame against such a permutation of a R-P-frame, we compute the following scores :

```
% compare modes
If modes (S-P-frame) is subsumed in modes (R-P-frame)
then add 0 else 4.

% compare types
If types (S-P-frame) is subsumed in types (R-P-frame)
then add 0 else 4.

% compare clauses
Score is sum of scores of best matches of each and every student clause to a reference clause
```


% compare clause ordering

If the *i*th clause of SPD is best matched to the *m*th clause of RPD, and the *j*th clause of SPD is best matched to the *n*th clause of RPD, and if the *m*th clause of RPD must precede the *n*th clause of RPD and the *i*th clause of SPD does not precede the *j*th clause of SPD, then add 2.

Student clauses are matched against canonical representations of reference clauses. To illustrate the problem of matching clauses, suppose APROPOS2 is matching the subgoals which appear before the recursive subgoals in the clauses which we call *prefix* subgoals :

```
fringe_length([X|Xs], N) :-  
    nonemptylist(X),  
    islist(X),  
    fringe_length(Xs, N1),  
    N is N1 + 1.
```

with the canonical reference clause :

```
fringe_length([X|Xs], N) :-  
    atom(X),  
    X \= [],  
    fringe_length(Xs, N1),  
    succ(N1, N).
```

then several mappings are possible. Each of "nonemptylist(X)" and "islist(X)" can be matched to "atom(X)" or "X \= []" or NOMATCH. NOMATCH means a redundant subgoal. Or both together can be matched to one of the reference prefix subgoals leaving the other reference prefix subgoal matched to NOMATCH. Or all can be matched to NOMATCH which would probably suggest the reference clause is not a good candidate to match with the student one.

One heuristic to make the search manageable is to first match student recursive clauses with reference recursive clauses, and then student base and nonrecursive clauses with reference base and nonrecursive clauses. Matching the recursive clauses first anchors the search as it specifies the SPD/reference-task mappings earlier than what later matching of the base and nonrecursive clauses can suggest. Before each matching of a pair of clauses, APROPOS2 has a record of the predicate/reference-task pairings which have been suggested. Each pair of clauses' matching will add to this record of the predicate/reference-task pairings. If the match suggests predicate/reference-task mappings which are not consistent with these predicate pairings, then further penalty scores are incurred. We order the list of SUGGESTED UNMAPPED SPD/TASK MAPPINGS, so APROPOS2 will attempt matching what gets put there first.

In matching recursive program clauses, the heuristic we use is to match the head goal first, then the recursive subgoal(s), then the *prefix* subgoals which appear before the recursive subgoal, and the subgoals which appear after the recursive subgoal which we call *suffix*

subgoals. When mapping student subgoals with reference subgoals, APROPOS2 maps student predicates to reference predicates and incurs penalties when a student predicate is mapped to more than one reference predicate or cannot be mapped consistently and vice versa. Similarly, APROPOS2 maps student variables to reference variables. The order is important since when a student term is mapped to a reference term, APROPOS2 applies Prolog unification. Unification will instantiate other terms and variables in the reference clause which are not yet mapped and these instantiations will affect subsequent matching. Here are some of the scores that contribute to matching clauses :

% compare clauses

Score of match between a pair of clauses is sum of scores of match of head goals, match of recursive goals, match of prefix goals and match of suffix goals.

% missing reference clause

If reference clause is matched to NOMATCH, ie., it has no other match with a student clause with a lower score, add 15.

% extra student clause

If student clause is matched to NOMATCH, ie., it has no other match with a reference clause with a lower score, add 15.

Here are some of the scores that contribute to matching a pair of clauses :

% equivalent subgoals

If student subgoal(s) is equivalent to reference subgoal(s), add 0.

% student subgoal and reference subgoal are different system predicates
Score of match is sum of scores of match of respective arguments plus 2.

% subgoals with same arities

Score of match is sum of scores of match of respective arguments.

% subgoals with difference of arity 1

Score of match is sum of scores of match of respective arguments plus 3.

% subgoals with difference of arity 2 or more

Score of match is sum of scores of match of respective arguments plus 7.

% missing reference subgoal

If reference subgoal is matched to NOMATCH, ie., it has no other match with a student subgoal with a lower score, add 8.

% extra student subgoal

If student subgoal is matched to NOMATCH, ie., it has no other match with a reference subgoal with a lower score, add 8.

% inconsistent subgoal matches

If the student subgoal predicate has been previously matched to a different reference subgoal predicate, or vice versa, keep record of both matches and add 2.

% side-effecting student subgoal

If student subgoal is a side-effect predicate like *write/1*, match it to NOMATCH and add 0.

In matching subgoals, APROPOS2 makes use of a database of Prolog knowledge to ensure an equivalent match is not missed. These facts are used as program transforms by rewriting the reference subgoals to more closely match the student's. This feature of the code-matching allows some parsimony in the representation of the task solutions. The facts concern properties of certain operators such as :

```

% commutative operators
commutative(=).
commutative(==).
commutative(\=).
commutative(+).
...

% associative operators
associative(+).
...

% equivalent(Subgoal1, Subgoal2)
% true if Subgoal1 is equivalent to Subgoal2
equivalent(X>Y, Y<X).
equivalent(X>=Y, Y<=X).
equivalent((X>=Y, Y>=X), X==Y).
equivalent((X<=Y, Y<=X), X==Y).
equivalent(succ(X, Y), Y is X + 1).
equivalent(X\=Y, \+(X=Y)).
...
equivalent(not(X), \+(X)).
equivalent(var(X), not(nonvar(X))).
equivalent(atomic(X), (atom(X); number(X))).           % it is unlikely that student will use
                                                         % a database reference

equivalent(number(X), (integer(X); real(X))).
...

```

If some equivalences are missed out, APROPOS2 in the critique phase will invoke the dynamic analysis module to show equivalence. It is possible to leave such reasoning about computational equivalence to this later phase for dynamic analysis (to the Prolog interpreter), but for efficiency sake, we compile some of these knowledge into such facts. Indeed, dynamic analysis only provides a weaker approach as it is only sound in the sense that it will always accept actual equivalent code as equivalent.

Here are some of the scores that contribute to the matching of arguments :

```

Let student's predicate argument be SA and reference argument be RA.
If SA == RA OR equivalent(SA, RA), add 0.

If (SA is an anonymous variable and var(RA)) OR (nonvar(SA) and var(RA)) OR (var(SA) and
nonvar(RA)), add 3.

If (list(SA) and not(list(RA))) OR (list(RA) and not(list(SA))), add 3.

If length(SA) \= length(RA), add 2.

```

If atomic(SA) and atomic(RA) and not(unify(SA, RA)), add 2.

If structures of SA and RA are lists, then score is sum of scores of recursively matching head and tail arguments.

If none of the above applies, unify(SA, RA) and add 0.

To illustrate an example, suppose APROPOS2 is matching variables in the student clause :

```
atom_count([_|Tail], N) :-  
    atom(X),  
    atom_count(Tail, N1),  
    N is 1 + N1.
```

with the variables in the canonical reference clause :

```
atom_count([H|T], N) :-  
    atom(H),  
    atom_count(T, N1),  
    succ(N1, N).
```

Here H in the canonical reference clause can be matched either to the anonymous variable or the variable X, but it is X we want. APROPOS2 maps H to the anonymous variable with a penalty score of 3 while it maps H to X both being variables with a penalty score of 0.

The penalty scores are obtained by considering how much of a misfit each match can possibly contribute to the difference of the mismatch, and has been tuned by testing the matching algorithm on an initial set of collected student reverse/2 programs. The values of the penalty scores have been constrained by the hierarchy of levels of impact on the mismatch. The scoring method is not fool-proof, but our empirical tests give us optimism that it will work reasonably well. The main reason seems to be that one code mismatch will get penalised in different feature matchings so the best match(es) is usually found if it exists. As part of future work on APROPOS2, we propose to study how sensitive is APROPOS2's performance to the choices of the scores. Code-matching with the use of a heuristic scoring function of closeness of match is no worse than most other code-matching approaches without a heuristic scoring function and that have no better principled ways of selecting the best fits.

The result of the best-first search is the best match (or matches) of the student predicate definitions with a set of reference predicate definitions. Each best match is thus a possible interpretation of the program. Each different interpretation will suggest a different set of fixes to a buggy student's program to make it work. APROPOS2 then goes into the critique phase in which a program critic criticises each match. It tries to explain differences in matching by dynamic analysis. Differences that cannot be explained away by dynamic analysis will be interpreted as bugs. If the student predicate definition is buggy and has been matched to a reference predicate definition, then bug corrections can be suggested.

5.4 Matching Permutations of Predicate Arguments

To illustrate the problem of matching permutations of predicate arguments, consider this clause in the insertion sorting/2 program :

```
sorting([X|Xs], Ys) :-  
    sorting(Xs, Zs),  
    insert(X, Zs, Ys).
```

This would require a predicate definition for insert/3 which should run for

```
insert(Element-to-be-inserted, List-to-insert-element-into, List-with-inserted-element).
```

The reference sorting/2 and insert/3 canonical programs should match as equivalent code to variants of the same program but with the the insert goals permuted in some consistent way. Thus when APROPOS2 matches the SPD for insert/3 with the RPD for insert/3, it tries all the various possible permutations, and selects the one match which has the lowest score.

One observation arises : can we use mode and type information to try only those permutations whose type and mode declarations agree with those of the invocation goal? Consider this buggy sorting/2 program

```
sorting([X|Xs], Ys) :-  
    sorting(Xs, Zs),  
    insert(X, Zs, Ys).  
sorting([], []).  
  
insert([], X, [X]).  
insert([Y|Ys], X, [Y|Zs]) :-  
    X < Y,  
    insert(Ys, X, Zs).  
insert([Y|Ys], X, [X, Y|Zs]) :-  
    X >= Y.
```

From the SPD for insert/3, APROPOS2 infers insert/3 has supportive type `insert(list(number), number, list(number))` and supportive mode `insert(+, +, -)`. The call invocation of insert in the sorting/2 program has type `insert(number, list(number), list(number))`. APROPOS2 infers the mode of the insert subgoal invocation in sorting/2 to be `insert(+, +, -)`.

Consider this insert/3 RPD :

```
insert(X, [], [X]).  
insert(X, [Y|Ys], [Y|Zs]) :-  
    X < Y,  
    insert(Ys, X, Zs).  
insert(X, [Y|Ys], [X, Y|Zs]) :-
```

$X \geq Y$.

which has supportive type $\text{insert}(\text{number}, \text{list}(\text{number}), \text{list}(\text{number}))$ and supportive mode $\text{insert}(+, +, -)$. This suggests that the SPD for $\text{insert}/3$ should match this RPD with the arguments (in RPD) permuted in order $[2,1,3]$ or $[3,1,2]$.

The $\text{insert}/3$ SPD is found to be best-matched with an argument permutation of $[2,1,3]$ of the $\text{insert}/3$ RPD. There are then two possible ways of fixing the student's program : to correct the subgoal invocation in $\text{sorting}/3$ to $\text{insert}(Ys, X, Zs)$ or to permute the insert arguments in the $\text{insert}/3$ predicate definition according to order $[2,1,3]$. Currently, we program APROPOS2 to choose only the first interpretation although it only needs a small modification to have APROPOS2 provide both interpretations which can be disambiguated by further analysis or simply asking the student.

Type and mode information may not always be useful if the student's program is incorrect. Consider this buggy $\text{insert}/3$ program :

```
insert([], [X], [X]).
insert([Y|Ys], [X], [Y|Zs]) :-
    X < Y,
    insert(Ys, [X], Zs).
insert([Y|Ys], [X], [X,Y|Ys]) :-
    X >= Y.
```

From the SPD for $\text{insert}/3$, APROPOS2 infers $\text{insert}/3$ needs supportive types $\text{insert}(\text{list}(\text{number}), \text{list}(\text{number}), \text{list}(\text{number}))$. This will fail to match the supportive types of all permutations of the $\text{insert}/3$ arguments of the $\text{insert}/3$ RPD.

Inferring supportive modes on a local basis will often provide the "?" mode for an argument position for a small Prolog procedure. This means not enough mode information can be inferred. Thus it is not clear how mode and type information can be always used to suggest possible permutations of arguments of a RPD that a SPD might match.

5.5 The Code-Critique Phase

In the code-critique phase, APROPOS2 fetches the winning node(s) that result from the best-first search. If there is only one winning node, this means that one best interpretation of the student's program has been found. More winning nodes mean alternative interpretations of the student's program. In our empirical experiments with APROPOS2, we have adopted the strategy of choosing the node(s) with the lowest score as the winner(s). In a later stage of PITS development when we consider student modelling, we can adopt a less strict strategy of choosing the node(s) with the lowest scores *together* with any other node(s) with scores that

vary from the lowest score by a small fixed margin. This offers the top few candidates as alternative program interpretations which may be further disambiguated by a student modelling component.

As APROPOS2 goes through each step of the code-critique phase, it will record any critique it finds. Only at the end of the code-critique phase will these critiques be printed out alongside the specific code they comment upon. The main information processed by the code-critic is the mappings of SPDs to RPDs. This is the information gathered from the best match in the code-matching phase, including the best permutations of the arguments of the head predicate for every matched RPD.

APROPOS2 criticises the main task SPD matched to a RP in the following way :

- For each match of a SPD to NOMATCH, APROPOS2 records a critique of an extra SPD.
- For each match of a RPD to NOMATCH, APROPOS2 records a critique of a missing SPD.
- For each match of a SPD to a RPD, the critique process is :
 - (1) Permute the reference subgoals' arguments in RPD accordingly using the list of matched-RPD/argument-permutation-order. Section 5.4 explains the reason for doing this. To illustrate, suppose the list of matched-RPD/argument-permutation-order from APROPOS2's analysis of a student's insertion-sort program is "sorting/2"/[1,2] and "insert/3"/[2,1,3]. Then before criticising the match of SPD sorting/2 to its RPD, APROPOS2 will permute the argument order of the insert subgoal in the RPD according to order [2,1,3] to ensure a correct consistent overall program interpretation is being matched against.
 - (2) Check record of results of dynamic analysis done in the earlier matches of SPDs to RPDs. If
 - there is a record of equivalence of function between SPD and RPD which means that the student predicate in the SPD has been found earlier to be equivalent to the reference predicate in the RPD, and
 - if the student predicate is directly or indirectly invoked *only* in another SPD whose predicate definition has been found earlier to be equivalent to a reference predicate,

then stop further work on criticising the syntactic closeness of fit between SPD and RPD, since they are presumed to produce equivalent desired results when used in a certain way. With this approach, APROPOS2 may be able to accept novel or unanticipated algorithms or implementations of student auxiliary predicates. This explains why we use the heuristic of criticising the match of the top task SPD first,

followed by the matches of SPDs called by this SPD, and then followed by matches of SPDs called by these previous SPDs, etc.

(3) Check for incorrect clause ordering by looking for this condition :

- the *ith* clause of SPD is best matched to the *mth* clause of RPD, and
- the *jth* clause of SPD is best matched to the *nth* clause of RPD, and
- if the *mth* clause of RPD must precede the *nth* clause of RPD *but*
- the *ith* clause of SPD does not precede the *jth* clause of SPD.

For each occurrence of this, record critique of the instance of wrong clause order in SPD.

(4) For each student clause in SPD mapped to NOMATCH, record critique of extra clause.

For each reference clause in RPD mapped to NOMATCH, record critique of missing clause. For each pair of student and reference clauses matched, criticise the syntactic closeness of the match of the head goals, the recursive subgoals if any, and then the prefix and suffix subgoals. Record critique of equivalent subgoals, incorrect subgoals, missing subgoals or extra subgoals as appropriate. In addition, if there is no good syntactic match between student and reference prefix or suffix subgoals, APROPOS2 will seek to prove their equivalence in function by running them on test data and checking if they produce identical results. The next section describes how this is done.

APROPOS2 judges a student program to be correct from its code-matching and critique process if :

- reference task algorithms have been matched to the student predicate definitions,
- matched implementations of these task algorithms are correct solutions, and
- any discrepancy in the match can be explained.

5.6 Dynamic Analysis of Programs

We use an example to show how the dynamic analyser of APROPOS2 performs its job of proving equivalence of function between student and reference predicates. Suppose APROPOS2 is criticising the match of the student prefix subgoal :

```
getlastrest(L, Last_El, Other_Els)
```

which appears in the second student clause of

```
reverse([], []).
reverse(L, [Last_El|Rev_Els]) :-
    getlastrest(L, Last_El, Other_Els),
    reverse(Other_Els, Rev_Els).
```

with the reference prefix subgoal of

```
append(List, [Last], Restlist)
```

which appears in the second reference clause of

```
reverse([], []).
reverse(List, [Last|Sofar]) :-
    append(RestList, [Last], List),
    reverse(RestList, Sofar).
```

When the getlastrest subgoal is mapped to the append subgoal, there is a syntactic mismatch of arguments. APROPOS2 tries to show equivalence by dynamic analysis. Earlier critique of the mapping of the head and recursive subgoals has produced this mapping of student variables to reference variables : L/List, Last_El/Last, Rev_Els/Sofar and Other_Els/RestList. The reference P-frame for the inverse naive reverse/2 has a test-cases slot with these information for running append/3 :

Invocation : append (X, [Y], [a]).	Result : append([], [a], [a]).
Invocation : append (X, [Y], [a,b]).	Result : append([a], [b], [a,b]).
Invocation : append (X, [Y], [a,b,c,d,e]).	Result : append([a,b,c,d], [e], [a,b,c,d,e]).

For each of this test case, using the list of variable mappings, APROPOS2's dynamic analyser creates an instance of an invocation of the student goal, in this example, getlastrest(L, Last_El, Other_Els), and instantiates L, Last_El and Other_Els to what their variable mappings are instantiated to when unified with the test case invocation. The dynamic analyser then runs the goal invocation of getlastrest(L, Last_El, Other_Els) on the student program and the goal invocation of append(RestList, [Last], List) on the reference program, and checks if they produce identical results. Two subgoals are deemed equivalent only if they produce identical results for all the test cases provided in the reference P-frame. To trap nonterminating computations which may result from a student buggy predicate definition,

we use a depth-bound on the goal stack size. If the depth bound is exceeded, then non-equivalence is presumed. If equivalence between the goal invocation of `getlastrest/3` and the goal invocation of `append/3` can be shown, then a record of the equivalence is kept so that further critique of the map of the `getlastrest/3` SPD to the `append/3` RPD will be deemed unnecessary as in this case. This is what gets recorded in APROPOS2. Ideally, we should also include mode and type information for the equivalence between `getlastrest/3` and `append/3` is only with respect to the mode of usage tested, that is, the mode and type of the invocation call. This should be a future extension to APROPOS2.

Essentially we have used a program testing approach to show equivalences when heuristic code-matching fails. As in program testing methodologies, the approach is not foolproof. To make program testing amenable, we have always attempted to cover the space of possible test data by breaking them down into equivalence classes and devising a suitable example of test data for each equivalence class. An interesting issue here which can be explored further is whether the generation of equivalence classes can be automated (possibly with the help of type information about the predicate) for a specific class of list-handling tasks. We can also explore the generation of test data from some specification of the task which may itself be in some Prolog representation. As correct implementations are never considered buggy, our program testing approach is conservative. This fits well with a pedagogical style that does not want to put the student off by criticising correct acceptable solutions as incorrect.

The above example shows dynamic analysis applied to comparing functionality between a pair of subgoals. Whenever there is syntactic code discrepancy between the student and the reference prefix or suffix subgoals, dynamic analysis is applied in the different possible ways. To illustrate, suppose the student prefix subgoals are

```
spred1({args}), spred2({args})
```

and the reference prefix subgoal is

```
rpred({args})
```

and there is syntactic mismatch between them, the dynamic analyser will first try to prove equivalence between `(spred1, spred2)` with `rpred`, failing which it will then try to prove equivalence between `spred1` and `rpred` (which may suggest the `spred2` subgoal is extra), and finally if the last attempt fails, between `spred2` and `rpred` (which may suggest the `spred1` subgoal is extra).

Figure 5-3 shows a schematic diagram of how APROPOS2 uses dynamic analysis to check for equivalences of student and reference predicates.

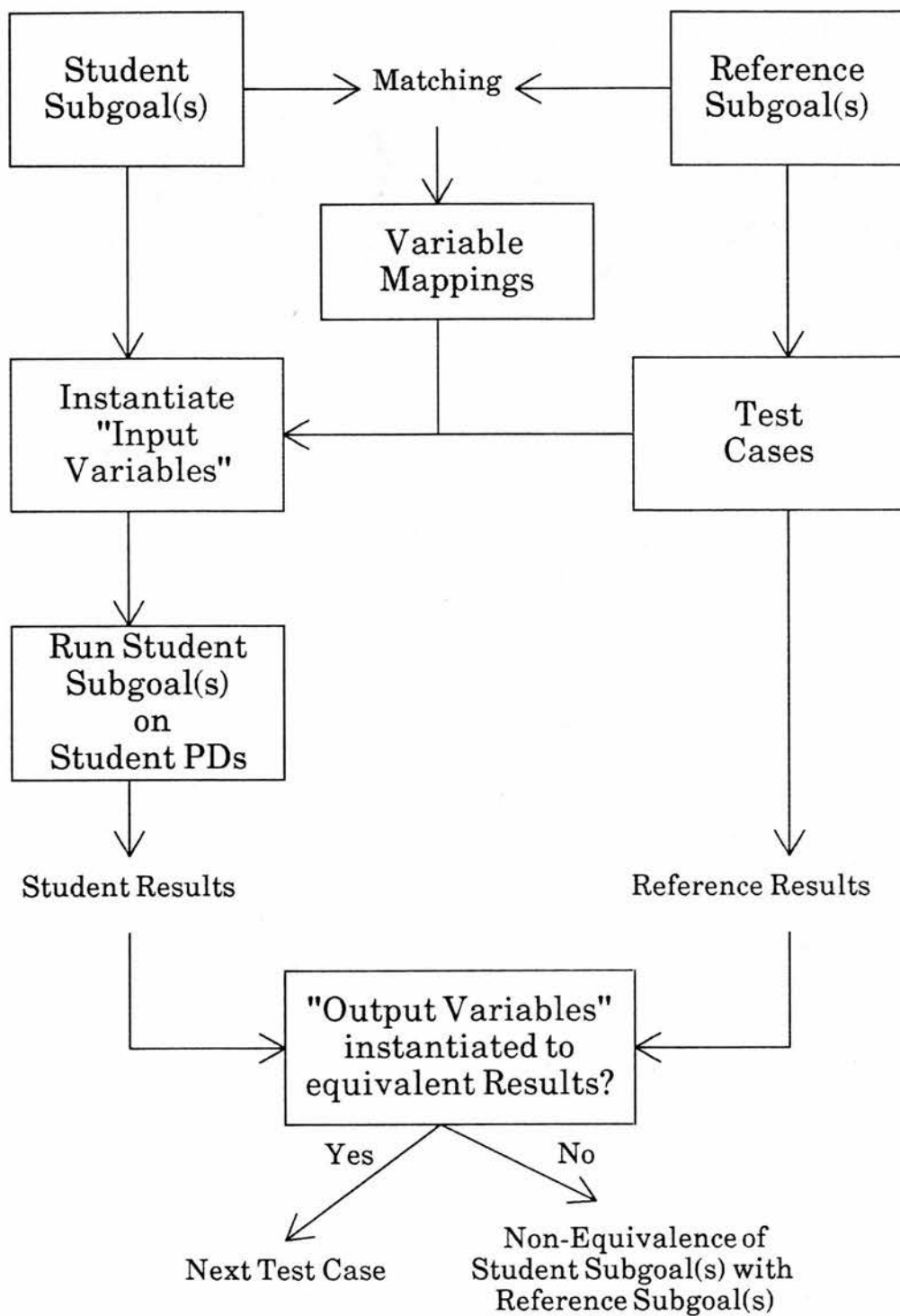


Figure 6-3 : Dynamic analysis in APROPOS2 to find equivalence of predicates

The code-matching and code-critique approaches of APROPOS2 insist on a fixed number of arguments for the main predicate of the task, but not for the auxiliary predicates. Each reference P-frame uses schemas to represent each clause. The representation of each subgoal is in its most commonly occurring forms (both correct and buggy) and is in terms of the necessary relationship between its arguments. Student prefix and suffix subgoals are first matched to permutations of these commonly occurring forms. If there is no match, this subgoal(s) is tested for equivalence with reference subgoal(s). Matched input arguments are instantiated to the same values and dynamic analysis is run to try to show their equivalence. This approach does not always work because, for example, uninstantiated arguments in the student subgoal which do not appear in the reference relation may affect the comparison, but it seems to be sound, that is, a correct student subgoal is always recognised as correct. If the student subgoal(s) is shown not to produce equivalent relations with the reference subgoal(s), then we can invoke Shapiro's program correction system (PDS6) to perturb it or its predicate definition to suggest corrections to fix the discrepancy. As described in Section 2.5.1, the perturbation is done based on assuming a bug equivalence class, that is, assuming a certain class of bugs is made, and fails if the bug(s) arises from several bug equivalence classes. Oracle queries are answered by relations computed from running matched reference predicates. We propose future work to consider how this aspect of debugging can be done by static analysis instead of dynamic analysis as it is currently being done now. One possible approach is to prove verification conditions following the success of a goal call. The difficulty with this approach is that in general it is difficult to synthesize verification conditions in the student's program.

5.7 A Detailed Case Study : sorting/2

5.7.1 Overview of Case Study

To illustrate APROPOS2's code-matching and code-critique process in more detail, we discuss a full case study, this time with APROPOS2 analysing the task of sorting a list. The task and student's solution are :

Task : SORTING A LIST

Write a Prolog program `sorting/2` which takes a list of numbers as input and sorts the numbers in the list in ascending order. A goal call "`sorting(X, Y)`" where `X` is the input list should succeed with `Y` instantiated to the sorted list. Use the predicate name `sorting` instead of `sort` which is an Edinburgh Prolog built-in predicate.

```
sorting([X|Xs], Ys) :-
    sorting(Xs, Zs),
    insert(X, Xs, Ys).
sorting([], []).

insert([], X, [X]).
insert([Y|Ys], X, [Y|Zs]) :-
    X < Y,
    insert(Ys, X, Zs).
insert([Y|Ys], X, [X,Y|Zs]) :-
    X >= Y.
```

Figure 5-4 : A buggy solution to `sorting/2`

The bugs in the student's program are :

- the `insert/3` subgoal in the recursive clause of `sorting/2` should be `insert(Zs, X, Y)`,
- the comparison test in the second clause of `insert/2` should be `X > Y`,
- the comparison test in the third clause of `insert/2` should be `X = < Y`,
- the head goal in the last `insert/3` clause should be `insert([Y|Ys], X, [X,Y|Ys])`.

The task-specific knowledge that APROPOS2 is provided for this sorting task are :

- the intended mode of use is `sorting(+, -)`
- both arguments of `sorting/2` must be of types `list(number)`
- representations of algorithms for sorting.

It infers a supportive mode of `insert(+, +, -)` for `insert/3`, and from the predicate definition of `insert/3`, that `insert/3` needs all its arguments to be of types `insert(list(number), number, list(number))`. APROPOS2 warns of singleton variables `Zs` in the first `sorting/2` clause, and `Ys` and `Zs` in the last `insert/2` clause.

APROPOS2 matches the student sorting P-frames with the reference P-frames for the various sorting algorithms – bubble sort, insertion sort, permutation sort and quick sort. It selects the best fit from the insertion sort P-frame. This fit suggests that the student's insert program should match the insert task. It finds that the best match for the student clause

```

sorting([X|Xs], Ys) :-
  sorting(Xs, Zs),
  insert(X, Xs, Ys).

```

is the canonical reference clause

```

sorting([X|Xs], Ys) :-
  sorting(Xs, Zs),
  insert(X, Zs, Ys).

```

with a discrepancy in the student insert subgoal.

Next, APROPOS2 matches the P-frame for the auxiliary predicate insert against the reference P-frames for the insert task. The reference P-frames for the insert/3 task are :

(a) the declarative version whose reference program is :

```

insert(X, [], [X]).
insert(X, [Y|Ys], [Y|Zs]) :-
  X > Y, (or X >= Y),
  insert(X, Ys, Zs).
insert(X, [Y|Zs], [X, Y|Zs]) :-
  X = < Y.

```

(b) the procedural version whose reference program is :

```

insert(X, [], [X]).
insert(X, [Y|Ys], [Y|Zs]) :-
  X > Y, (or X >= Y),
  !,
  insert(X, Ys, Zs).
insert(X, [Y|Zs], [X, Y|Zs]) . (or insert(X, Zs, [X|Zs])).

```

APROPOS2 finds the types of the insert predicate in the student's insert/3 to be `insert(list(number), number, list(number))` but the types of insert in these reference P-frames are `insert(number, list(number), list(number))`. It attempts to match the student's insert/3 with different permutations of the arguments of insert/3 in these P-frames. It discovers that permutation [2, 1, 3], that is, interchange the first 2 arguments, best matches the student insert/3 and is consistent with the type analysis. This match also reveals some bugs in the student's insert/3.

In the critique phase, APROPOS2 finds that there is a discrepancy between the insert subgoal `insert(X, Xs, Ys)` in `sorting/2` with the reference P-frame `insert(Zs, X, Ys)` which is

permuted from `insert(X, Zs, Ys)` from permutation [2, 1, 3]. It does a dynamic analysis but cannot show equivalence. This is the final bug report :

Your program seems to be incorrect.
Now, this is APROPOS2's critique on your program :

Clause 1 of `insert/3` seems fine.
This base case says that the result of inserting an element `X` into the empty list is the list `[X]`.

In clause 2 of `insert/3`, replace
`X < Y` with `X > Y`.
This clause handles the case when the element to be inserted is larger than the first element of the list to be inserted into.

In clause 3 of `insert/3`, replace
`insert([Y|Ys], X, [X, Y|Zs])` with `insert([Y|Ys], X, [X, Y|Ys])`
and
`X >= Y` with `X = < Y`.
This clause, which does the swapping, handles the case when the element to be inserted is NOT larger than the first element of the list to be inserted into.

In clause 1 of `sorting/2`, replace
`insert(X, Xs, Ys)` with `insert(Zs, X, Ys)`.
This recursive case says that to sort a list, first sort the tail of the list and then insert the first element of the original to the result of sorting the tail.

Clause 2 of `sorting/2` seems fine.
This base case says that the result of sorting the empty list is the empty list.

Figure 5-5 : APROPOS2's analysis of `sorting/2` program in Figure 5-4

One observation of APROPOS2's output is that the student has mixed up the number comparison tests operators in both the recursive clauses of `insert/3`. It will be useful for APROPOS2 to recognise the consistency of the bugs so that the bug report can be made more explicit concerning the student's misunderstanding. This might be done by having buggy versions of the `equivalent/2` matches in the database of Prolog knowledge used in the code-matching phrase, described in Section 5.3.

In this case study, APROPOS2 finds one best program interpretation as is usually the case in our empirical experiments. If there is more than one program interpretation, APROPOS2 will print out all of them. We intend that the student modelling component of PITS will take as input from APROPOS2 the best few program interpretations for further processing.

Since the algorithm and its decomposition into predicate definitions, clauses, subgoals, arguments and terms have been found, APROPOS2 can print out more of this information than is currently being done. Although it is relatively easy to incorporate this in APROPOS2, printing pertinent information and not presenting the student with too much information, some of which he may already know, is a matter for further research in student modelling and tutoring strategy.

5.7.2 Details of Case Study

Figure 5-6 shows the best-first search that results from the code-matching process of APROPOS2 in working on the student program in Figure 5-4. Node 1 is the initial node. In the first step, the sorting/2 SPD is mapped to the RPDs for bubble-sort, insertion-sort, permutation-sort and quick-sort resulting in Nodes 2, 3, 4 and 5. Node 6 is the result of mapping the SPD to NOMATCH. When all successors of Node 1 have been created, the next node to be expanded is the one with the lowest score, in this case, Node 3. Node 3 is expanded by mapping the SPD insert/3 to the reference task insert/3 which is suggested by the matching done in node 3. The SPD insert/3 is mapped to the possible RPDs for the reference task insert/3, resulting in Nodes 7 and 8. Node 9 is the result of mapping the SPD to NOMATCH. Node 7 is now the node with the lowest score, and the search terminates since it is a complete mapping.

APROPOS2's code-matcher first creates this node :

-
- **NODE NUMBER** : 1
 - **NODE SCORE** : 0
 - **SPD/RPD MAPPINGS** : []
 - **UNMAPPED SPDs** : [sorting/2, insert/3]
 - **SUGGESTED UNMAPPED-SPD/TASK MAPPINGS** : [sorting/2/sorting/2]
-

The code-matcher expands this initial node by mapping the sorting SPD to the bubble-sort RPD. The score of this match is 36. This match suggests no mappings for the reference swap-elements/2 task and for the student insert/3. Thus the cost estimate to extend this partial mapping to a complete one is 50 since the student insert/3 will not be able to be mapped to any reference task.

-
- **NODE NUMBER** : 2
 - **NODE SCORE** : $g + h = 36 + 50 = 86$
 - **SPD/RPD MAPPINGS** : [sorting/2/bubble-sort/2]
 - **UNMAPPED SPDs** : [insert/3]
 - **SUGGESTED UNMAPPED-SPD/TASK MAPPINGS** : [[]/swap-elements/2, insert/3/[]]
-

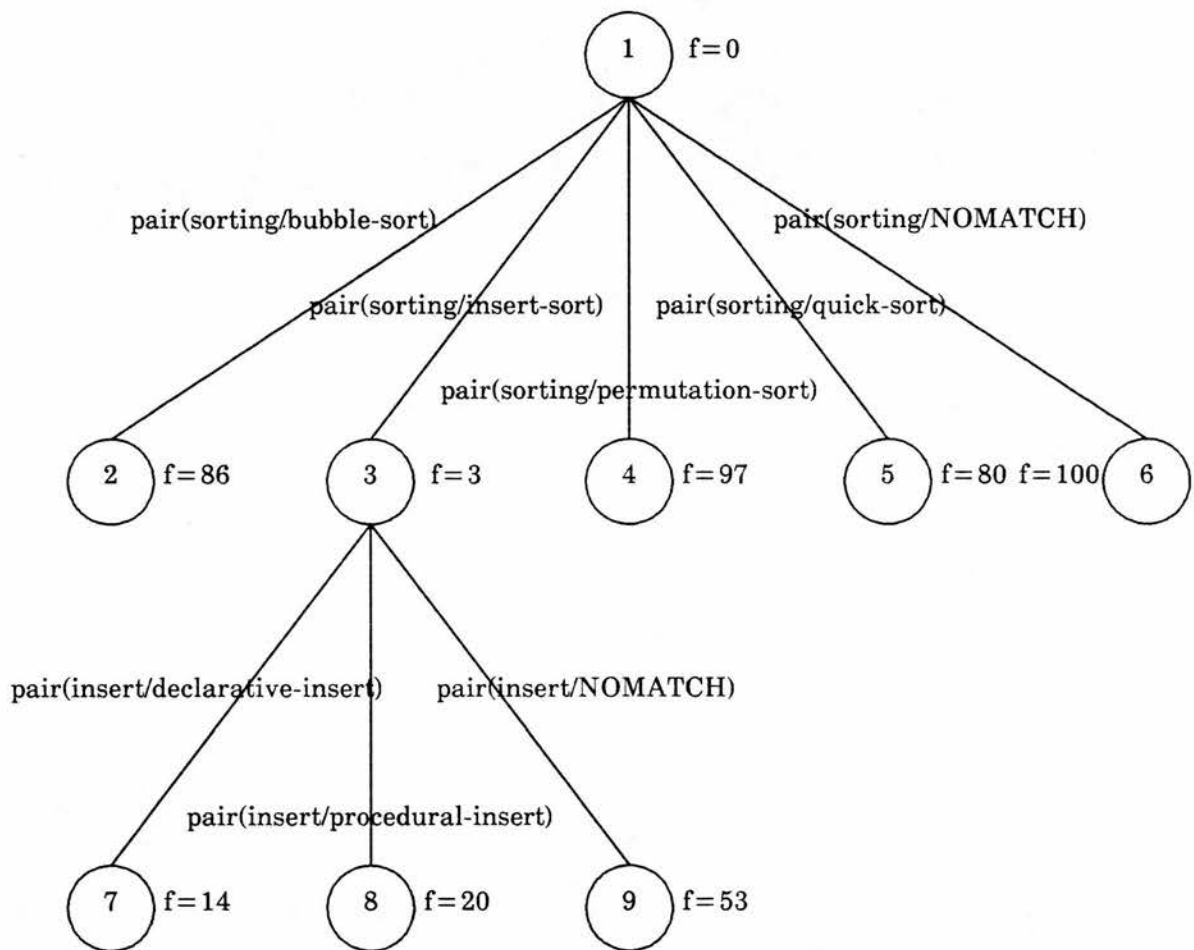


Figure 5-6 : Best-first search for sorting/2 program in Figure 5-4

In the next node creation, the code-matcher maps the sorting SPD to the insertion-sort RPD. The score of this match is 3. This match suggests the student insert/3 SPD is to be mapped to the reference insert/3 SPD. The cost estimate to extend this partial mapping to a complete one is the conservative score of 0 which never overestimates the actual estimate.

-
- **NODE NUMBER** : 3
 - **NODE SCORE** : $3 + 0 = 3$
 - **SPD/RPD MAPPINGS** : [sorting/2/insertion-sort/2]
 - **UNMAPPED SPDs** : [insert/3]
 - **SUGGESTED UNMAPPED-SPD/TASK MAPPINGS** : [insert/3/insert/3]
-

In the next node expansion of Node 1, the code-matcher maps the sorting SPD to the permutation-sort RPD. The score of this match is 47. This match suggests no mappings for the reference ordered/1 and permutation/2 tasks and for the student insert/3. The cost estimate to extend this partial mapping to a complete one is 50.

-
- **NODE NUMBER** : 4
 - **NODE SCORE** : $47 + 50 = 97$
 - **SPD/RPD MAPPINGS** : [sorting/2/permutation-sort/2]
 - **UNMAPPED SPDs** : [insert/3]
 - **SUGGESTED UNMAPPED-SPD/TASK MAPPINGS** : [[]/ordered/1, []/permutation/2, insert/3/[]]
-

In the next node creation, the code-matcher maps the sorting SPD to the quick-sort RPD. The score of this match is 30. This match suggests no mappings for the reference partition/4 and append/3 tasks and for the student insert/3. The cost estimate to extend this partial mapping to a complete one is 50.

-
- **NODE NUMBER** : 5
 - **NODE SCORE** : $30 + 50 = 80$
 - **SPD/RPD MAPPINGS** : [sorting/2/quick-sort/2]
 - **UNMAPPED SPDs** : [insert/3]
 - **SUGGESTED UNMAPPED-SPD/TASK MAPPINGS** : [[]/partition/4, []/append/3, insert/3/[]]
-

In the last node expansion of Node 1, the code-matcher maps the sorting/2 to NOMATCH. The score is 50. If this node wins, this implies that APROPOS2, with what knowledge of the task it has, finds the SPD incomprehensible. This means that algorithm recognition has failed. The possible reasons are :

- APROPOS2 has no representation of the algorithm the student is attempting

- APROPOS2 has no representation of the implementation of the algorithm the student is attempting
- APROPOS2 has the representation of the implementation of the algorithm the student is attempting, but the student code differs much from what APROPOS2 expects from a program that implements the algorithm.

The cost estimate to extend this partial mapping to a complete one is 50.

-
- **NODE NUMBER** : 6
 - **NODE SCORE** : $50 + 50 = 100$
 - **SPD/RPD MAPPINGS** : *[sorting/2/NOMATCH]*
 - **UNMAPPED SPDs** : *[insert/3]*
 - **SUGGESTED UNMAPPED-SPD/TASK MAPPINGS** : *[insert/3/[]]*
-

When all possible successors for Node 1 has been created, the code-matcher looks at the list of active open nodes to find one with the lowest score. It finds Node 3 and as Node 3 is not a complete mapping yet, the code-matcher now proceeds to expand it. From the list of **SUGGESTED UNMAPPED-SPD/TASK MAPPINGS** in Node 3, it knows that the student *insert/3* is to be mapped to the reference task *insert/3*. It fetches all the reference P-frames for *insert/3* and matches the student *insert/3* P-frame against them. The resulting nodes are :

-
- **NODE NUMBER** : 7
 - **NODE SCORE** : 14
 - **SPD/RPD MAPPINGS** : *[sorting/2/insertion-sort/2, insert/3/declarative-insert/3]*
 - **UNMAPPED SPDs** : *[]*
 - **SUGGESTED UNMAPPED-SPD/TASK MAPPINGS** : *[]*
-

-
- **NODE NUMBER** : 8
 - **NODE SCORE** : 20
 - **SPD/RPD MAPPINGS** : *[sorting/2/insertion-sort/2, insert/3/procedural-insert/3]*
 - **UNMAPPED SPDs** : *[]*
 - **SUGGESTED UNMAPPED-SPD/TASK MAPPINGS** : *[]*
-

-
- **NODE NUMBER** : 9
 - **NODE SCORE** : 53
 - **SPD/RPD MAPPINGS** : *[sorting/2/insertion-sort/2, insert/3/NOMATCH]*
 - **UNMAPPED SPDs** : *[]*
 - **SUGGESTED UNMAPPED-SPD/TASK MAPPINGS** : *[]*
-

At this stage, all successors of Node 3 have been created. The code-matcher next fetches the active node with the lowest score. It finds Node 7, and since Node 7 has no unmapped SPDs, the search terminates.

We now discuss the creation of Node 7 in more detail. The student insert/3 P-frame is being matched against the reference declarative-insert/3 P-frame. There are $3! = 6$ permutations of arguments of the insert goal so there are 6 versions of the canonical reference program to be matched against. The next figure shows the result of matching the student P-frame against permutation [1, 2, 3] of the reference P-frame. The score of 28 for matching clauses is contributed by several mismatches in the code.



Figure 5-7 : Match of student insert/3 P-frame with reference declarative-insert/3 P-frame with argument permutation [1, 2, 3]

The student P-frame of insert/3 is also matched against permutations [1,3,2], [2,1,3], [2,3,1], [3,1,2], and [3,2,1] of the reference P-frame with corresponding scores of 32, 11, 32, 24 and 36. Figure 5-8 shows the result of matching the student P-frame against permutation [2, 1, 3] of the reference P-frame which offers the best fit of the permutations. The score of 7 for matching clauses is contributed from the mismatch of the prefix subgoals.

Matching student's insert/3 with reference's declarative-insert/3 of algorithm insert/3 ...

Trying permutation order [2, 1, 3]

Score for matching predicate name : 0
 Score for matching number of arguments : 0
 Score for matching number of clauses : 0
 Score for matching modes : 0
 Score for matching types : 4
 Score for matching clause ordering : 0
 Score for matching clauses : 7

Match of clauses :

Student Clause :	Reference Clause Matched :
Clause No : clause(2)	Clause No : ref-clause(2)
Clause Type : recursive	Clause Type : recursive
Head : insert([Y Ys],X,[Y Zs])	Head : insert([Y Ys],X,[Y Zs])
Prefix : X < Y	Prefix : X > Y
Recursive : insert(Ys, X, Zs)	Recursive : insert(Ys, X, Zs)
Suffix : []	Suffix : []
Student Clause :	Reference Clause Matched :
Clause No : clause(1)	Clause No : ref-clause(1)
Clause Type : base	Clause Type : base
Head : insert([], X, [X])	Head : insert([], X, [X])
Prefix : []	Prefix : []
Recursive : []	Recursive : []
Suffix : []	Suffix : []
Student Clause :	Reference Clause Matched :
Clause No : clause(3)	Clause No : ref-clause(3)
Clause Type : nonrecursive	Clause Type : nonrecursive
Head : insert([Y Ys],X,[X,Y Zs])	Head : insert([Y Zs],X,[X,Y Ys])
Prefix : X > = Y	Prefix : X = < Y
Recursive : []	Recursive : []
Suffix : []	Suffix : []

Total Score of match : 11

Figure 5-8 : Match of student insert/3 P-frame with reference declarative-insert/3 P-frame with argument permutation [2, 1, 3]

The best-match found at the end of APROPOS2's code-matching process is :

sorting/2/insertion-sort/2 with argument permutation [1,2],

insert/3/declarative-insert/ 3 with argument permutation [2,1,3]

with the following discrepancies :

In clause 2 of insert/3,
X < Y with reference's X > Y.

In clause 3 of insert/3,
insert([Y|Ys], X, [X, Y|Zs]) with reference's insert([Y|Ys], X, [X, Y|Ys])
and
X >= Y with X = < Y.

In clause 1 of sorting/2,
insert(X, Xs, Ys) with reference's insert(Zs, X, Ys).

In the code-critique phase, APROPOS2 will try to account for these discrepancies. In this case, it fails and all these discrepancies will be treated as bugs and presented to the student.

5.8 Summary

The code-matching approach of APROPOS2 relies on information from several sources of expertise to match Prolog programs. It uses heuristic methods to recognise different algorithms and implementations of algorithms, and a program testing approach to reason about computational equivalence of program fragments.

Our debugging approach is related to the debugging approach of Murray [Murray 85] which uses formal methods (the Boyer-Moore theorem prover) to reason about computational equivalence. Murray has proposed a similar approach to debugging Prolog programs using Shapiro's diagnosis routines but he converts the Prolog program first to its LISP equivalent which poses another problem in itself.

Our approach is also related to the program interpretation approach of Johnson [Johnson 85] which uses heuristics to put goals onto a agenda and selects plans which implement a goal and matches these plans to the student Pascal code. We use heuristics to constrain our program interpretation space in APROPOS2. For example, the top-level predicate definition is matched first. This match suggests putting new tasks onto a goal agenda, so to speak, for subsequent matching against a representation of the student's program. APROPOS2's goals are the tasks which are invoked in the reference program matched to the student program. PROUST always finds just one best interpretation to proceed and then sticks to it. It cannot backtrack to another alternative as it does not keep a function of the suitability of each match. In APROPOS2, we use a best-first search to select the best fit(s), thus ensuring that the best match(es) if it exists can always be found.

The Programmer's Apprentice (PA) Project [Rich 81] parses a LISP program to recognise the plans used in it by using a library of several commonly used correct plans. It would parse different implementations of the same algorithm into quite different plan representations if

their control and data flow differ. It is designed to parse correct program implementations and requires exact matching of flow graphs. It would be difficult for the PA parser to recognise algorithms in buggy student programs since discrepancies in graph matches will be attributed to bugs and the partial matching required would greatly increase the number of possible matches between graphs. Unlike APROPOS2, the PA does not use a measure of mismatch. Furthermore, it does not reason with semantics of programming constructs to test for computationally equivalent fragments of code.

In summary, we have described the code-matching process of APROPOS2 which entails a best-first search to find the best mappings of student predicate definitions to reference predicate definitions, and the code-critique process in which APROPOS2 tries to account for any discrepancy between the student code and the correct code. We have also presented a case study in some detail to illustrate APROPOS2's code-matching process of algorithm recognition, bug detection and bug correction.

Chapter 6. Development of APROPOS

The system we have described so far in this thesis is the current version of APROPOS (APROPOS2). Two key features of APROPOS2 are :

- a general code-matching process which is independent of the task knowledge, and
- a best-first search for the best matches of representations of student programs to representations of reference programs.

We will further argue why these features are critical for APROPOS2's performance and how they address the limitations of earlier versions of APROPOS which did not incorporate these features. In this chapter, we discuss the initial development of APROPOS. A first version of APROPOS (APROPOS1) was developed to experiment with some initial ideas for integrating dynamic analysis with static analysis [Looi & Ross 87]. A second version of APROPOS (APROPOS-AS) was developed to explore an analysis-by-synthesis approach for analysing failure-driven loops in Prolog. In this chapter, we describe the debugging approaches in these systems, discuss some empirical work we have carried out using these systems, and highlight their limitations and the lessons learned for APROPOS2.

6.1 Chronological Development of APROPOS

Figure 6-1 shows the chronological development of the various APROPOS systems. APROPOS1 was first developed to experiment with some initial ideas for integrating dynamic analysis with static analysis. APROPOS-AS was developed at about the same time to explore an analysis-by-synthesis approach for analysing a task requiring the use of a failure-driven loop. To test APROPOS1 on the task of list-reversal, we ran it on a set of hand-written solutions collected from the second year Artificial Intelligence (AI-2) class in the Department of Artificial Intelligence, University of Edinburgh (Data Set 1). We enlisted volunteers from this same class and the Master of Science (MSc) class of 1986/87 in the department to try

- APROPOS1 interactively, collecting Data Sets 2 and 3 respectively, and
- APROPOS-AS interactively, collecting Data Sets 4 and 5 respectively.

We evaluated the performance of APROPOS1 and APROPOS-AS on these data sets. Further details of the experiments conducted and the evaluation will be discussed in Sections 6.3 and 6.5.

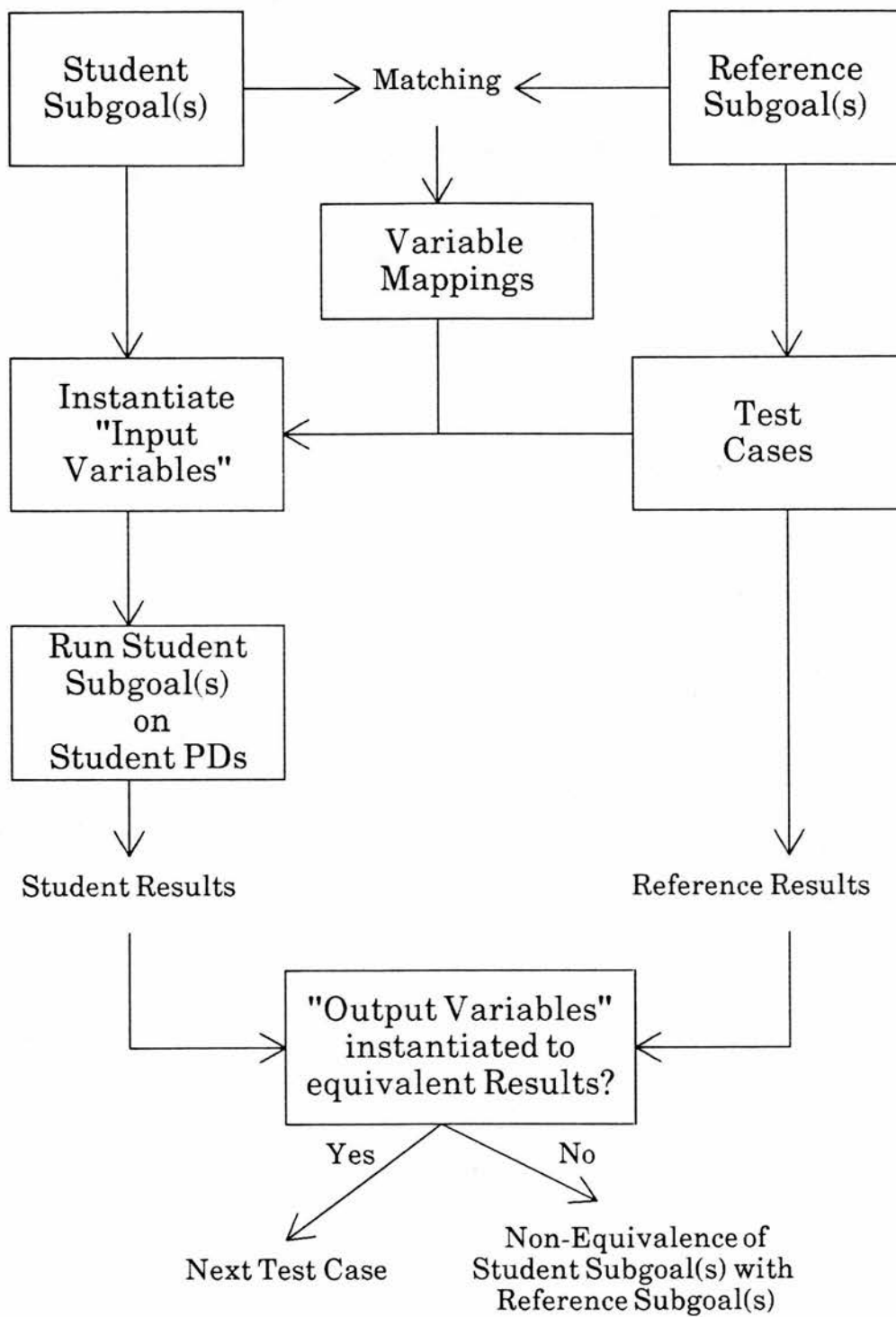


Figure 5-3 : Dynamic analysis in APROPOS2 to find equivalence of predicates

Consequent to all these efforts, we developed APROPOS2 drawing on the lessons we learnt. We tested APROPOS2 on the corpus of students' programs for `reverse/2` collected in the experiments for APROPOS1 (Data Sets 1, 2 and 3) and on a new corpus of programs for other tasks collected from the MSc class of 1986/87 in the the department (Data Set 6).

6.2 Feature Extraction in APROPOS1

Unlike APROPOS2, APROPOS1 does no best-first search; instead, it extracts features such as the use of the iteration or recursion arguments, and the use of an accumulator argument from the code. The extraction of these techniques has been described in Section 3.6. APROPOS1 uses these features to recognise which algorithm may have been intended in the student's program. For the `reverse/2` task, the use of three arguments – including an iteration argument and an accumulator argument which "accumulates" in the recursive subgoal – suggests that the railway-shunt algorithm is intended. The use of two arguments – an iteration argument in a '+' mode and a result argument in a '-' mode – suggests that the naive algorithm is intended. The use of two arguments – an iteration argument in a '-' mode and different terms in the other argument in a '+' mode – suggests that the inverse naive algorithm is intended.

One correct reference program is chosen as the best match, and this provides mapping between predicate and variable names. If there is a discrepancy in the match, then an attempt is made to explain it first by fitting it to commonly occurring bug prototypes. If the difference cannot be explained away by bugs, that part of the student code will be run on examples. To trap non-terminating computations, a bound is set on the size of the stack of goals. The execution results are compared against results returned by running the same examples on the reference program. If there are errors, Shapiro's bug detection strategy can be used to further localise the error. If it needs to ask oracle queries about program divergence, attempts will be made to answer them by using well-founded goal-ordering relations to the successive predicate goal calls. For example, if in list reversal, `reverse([H/T], Result1)` and `reverse(T, Result2)` are successive calls in the recursion, then the list-length of `Result1` must be longer than the list-length of `Result2`. Since we cannot depend on Shapiro's bug correction routines to correct the error as it relies on too many queries, the reference predicate definitions are used to suggest corrections for bugs. Unmatchable goals are handled by adding reference clauses that match the goals.

APROPOS1 was constructed to handle a class of recursive tasks, like `reverse/2`, `length/2` and `factorial/2`. The programs for these tasks usually have a common structure in which there is a recursive subgoal call that works on a substructure from the parent goal. APROPOS1 searches for the recursive subgoal in the body of a clause. It looks for the iteration and the result terms in the head and recursive goals. If there is a recursive subgoal, it looks for

standard cliches in the subgoals that appear before and after the recursive subgoal in the body. For reverse/2, it expects to find *append*(_, [_], _) after the recursive subgoal. If APROPOS1 finds it before the recursive subgoal (which gives an inefficient reverse/2 with fatal backtracking behaviour when used conventionally), then it will comment on the efficiency of the program. If it does not find any cliché, then it will look for the subgoals that relate the result term in the head goal to the result term in the recursive goal. It symbolically executes the subgoals it can find for test inputs, and compares the instantiated results with those that are obtained by executing the standard cliché. If the student's program is incorrect, it will fail to produce correct results for these test cases. If the error is not already localised, Shapiro's diagnosis routines can be used to localise it. If the student's program is correct, it will succeed on these test cases. Using this approach, correct implementations are never considered incorrect. When the student uses a novel algorithm or when APROPOS1 fails to understand the student's program, it will run it for a number of test cases using Shapiro's diagnosis routines. Ideally, a more rigorous approach for proving equivalence formally is better, but future work needs to be done to bring it into practical use.

Figure 6-2 shows the main task information used by APROPOS1 for analysing the naive algorithm for reverse/2. In analysing the recursive clause, APROPOS1 tries to detect the iteration and the result arguments in the student's program and matches them against the expected arguments of a correct program. The explanation capability of APROPOS1 comes from the template commentaries which are attached to common correct and buggy prototypes of these arguments and the other subgoals which inter-relate these arguments. Several base clauses are used in order to check whether the student might have written redundant clauses.

Task name :	reverse/2
Algorithm used :	naive reverse
Types :	both arguments are lists
Modes :	reverse(+, -), determinate, total
Number of clauses :	2
Base Clauses :	reverse([], []), reverse([A], [A]), reverse([A, B], [B, A])
Recursive Clause :	<div> Iteration Arg in Head Goal : [H T] Iteration Arg in Recursive Subgoal : T Result Arg in Head Goal : Res Result Arg in Recursive Subgoal : Sofar Relation between Sofar and Result : append(Sofar, [H], Res) (with link to the task representation for append/3) Constraint : append subgoal to come before recursive subgoal </div>
Several Test Cases	for trying out reverse/2 and append/3

Figure 6-2 : Main task information for naive reverse/2

6.3 Empirical Evaluation of APROPOS1

In this section, we discuss the performance of APROPOS1 in debugging student solutions to the reverse/2 task.

6.3.1 Set-Up of Evaluation of APROPOS1

Data Set 1 is a set of 49 hand-written solutions to the task of list-reversal collected from the second-year Artificial Intelligence (AI-2) class of 1986/87 in the Department of Artificial Intelligence, Edinburgh University. The students have covered Prolog syntax, basic control flow, AND/OR trees, recursion, lists and unification, but have not yet been assigned a programming assignment. The students were asked to write their programs in class after some two weeks of Prolog instruction. We ran APROPOS1 on 40 of these solutions which have no syntax errors. There were 9 programs which were syntactically incorrect and which we will not consider.

Data Set 2 is the complete protocols of 9 AI-2 students' interactive sessions with APROPOS1. These 9 students reported having no prior Prolog programming experience. They volunteered to try APROPOS outside of their classroom hours. Each student tried APROPOS1 on a SUN terminal. APROPOS1 provides a critique of a student's program if it is syntactically correct. The student is expected to read the advice and edit his program if it is incorrect. APROPOS1 will then analyse the edited program. Six of these students eventually managed to write a program for reverse/2 that returns a correct solution in the first run.

Data Set 3 is the complete protocols of 16 Master of Science, AI (MSc) student's interactive sessions with APROPOS1. They tried APROPOS1 at some time during their third and fourth week of Prolog instruction. All of them were able to arrive at a program for reverse/2 that returns a correct solution in the first run.

In total, 25 students from the Prolog classes of 1986/87 who tried APROPOS1 interactively, and 22 of these students were able to arrive at a first correct version of reverse/2 after an interactive session with APROPOS which criticises each syntactically correct program which the student produced. Two students floundered too much and were unable to write a correct program. One student aborted his session as he has not enough time to complete it. For many of them, it was their first Prolog programs they had written and executed on a computer.

6.3.2 Discussion of APROPOS1's Program Analyses

Empirical results show that APROPOS1 can detect a good proportion of bugs and can suggest corrections for many of these detected bugs. Appendix II shows a full session listing of

APROPOS1's analyses of a student's programs for the reverse/2 task. Figure 6-3 shows an example of APROPOS1's analysis. The student had earlier produced a correct railway-shunt reverse/2 which APROPOS1 accepted as correct, and had gone on to try another different program for reverse/2 :

Your program clauses are :

```
append([], L, L).
append([H], L, [H|L]) .
append([H|T], L, Ans) :-
    append(T, L, X),
    append([H], X, Ans).
reverse([], []).
reverse([H], [H]).
reverse([H|T], L) :-
    reverse(T, M),
    append(M, [H|L], L).
```

Your base clause of reverse/2 :

```
reverse([], []).
```

seems fine.

Let's run your program on an example : reverse([], X).

Your program produces solution : reverse([], []) which is correct, and no more solutions.

Your base clause of reverse/2 :

```
reverse([H], [H]).
```

seems redundant.

Analysing clause :

```
reverse([H|T], L) :-
    reverse(T, M),
    append(M, [H|L], L).
```

the subgoal append(M, [H|L], L) is not quite right. How about
append(M, [H], L).

The intention is that the first argument of reverse/2 will be a list, and that the second argument will be a variable to be instantiated to the reverse of that list or, of course, a list to be tested to see if it is the reverse of the first list. Thus

```
:- reverse([a,b,c], List).
```

should instantiate List to [c,b,a], and

```
:- reverse([a,b], [b,a]).
```

should succeed.

How can the answer [c,b,a] be derived from [a,b,c]? You know that in Prolog, if you are given the list [a,b,c], it is very easy to access the first element a and the tail [b,c]. How is [c,b,a] made out of these parts?

Clearly, you need to reverse [b,c] to get [c,b], and then you need to put a at the end. An easy way to do this is to append [c,b] and [a]. This approach is represented by the recursive clause. The base case is the test for the end of the recursion.

Figure 6-3 : An example of APROPOS1's analysis of a student's reverse/2

APROPOS1 applied dynamic analysis to find if an invocation of the student subgoal append(M , [H|L], L) might produce the same result for L as the invocation of the correct

append(M, [H], L). The first invocation resulted in a run that exceeded the goal stack size, so non-equivalence is deduced.

The latter part of APROPOS1's commentary – *"Analysing clause ... the recursion."* – will be output if APROPOS1 finds that the program does not have the correct append/3 subgoal. It is intended to assist the student to understand what is required to build up the desired result in the recursive clause.

The student subsequently modified the append subgoal in the reverse recursive clause to append(M, H, L), which APROPOS1 did not accept as correct. Eventually, the student used append(M, [H], L) which APROPOS1 accepted as correct.

In the analyses of some students' programs, there were a number of bugs which were not detected and commented upon. This is not because of limitations in the debugging approach but because of the incomplete coding of APROPOS1. Thus deficiencies in APROPOS1 are the cause for APROPOS1's failure to comment on these buggy clauses of different students' programs :

```
reverse([X|L], [X|L1], L2) :-  
    reverse(L, L1, L2).
```

```
reverse([H|T], L, [H,M]) :-  
    reverse(T, L, M).
```

```
reverse([Head|Tail], X) :-  
    append(Y, [Head], X),  
    reverse(Y, Tail).
```

Parts of APROPOS1's procedures are task-specific for the reverse/2 task. This makes it difficult to encode APROPOS1 to handle new tasks. But APROPOS1 demonstrates that the approach of integrating dynamic analysis to supplement code analysis is feasible. In APROPOS2, we seek to generalise the debugging algorithm, allow the task-specific information to be encoded separately, and distinguish clearly the code-matching from the code-critique phase. APROPOS2 generalises the debugging algorithm so that it can be used for a wider range of tasks. It also incorporates a best-first search to interpret the student program. As the code analysis process is independent of the task examined, it can be used to analyse a new task by encoding the task knowledge.

APROPOS1 fails in its analysis if it cannot recognise the iteration or the result variables. Thus it is unable to comment on this clause :

```
reverse([Head|Tail], [Contents_of_reversed_tail|[Head]]) :-  
    reverse(Tail, [Contents_of_reversed_tail]).
```

since it cannot detect the result variables which have the list structure [...]. This illustrates the limitation of the feature-extraction approach that picks out features from a student program and compares them with those of a correct reference program. Contrast this with a code-matching approach which compares code with code and thus can always criticise a match of two terms. APROPOS2 using the code-matching approach has this to say about this same clause :

Clause 3 of reverse/2 does not seem quite right.

Replace `reverse([Head|Tail], [Contents_of_reversed_tail|[Head]])` with

`reverse([Head|Tail], X),`
and `reverse(Tail, [Contents_of_reversed_tail])` with
`reverse(Tail, Y).`

and `append(Y, [Head], X)` seems to be missing.

This is what your clause should look like :

`reverse([Head|Tail], X) :-`
`reverse(Tail, Y),`
`append(Y, [Head], X).`

This recursive case says that the reverse of a non-empty list can be found by reversing the tail of the list, and then appending a list consisting only of the first element of the original to the result of reversing the tail.

Analysis of APROPOS1's output shows that its bug reports describe actual errors in the incorrect students' programs, although students' reactions to the advice given is mixed. After trying the interactive version of APROPOS1, some students have commented that in some cases, APROPOS1 could tell the code was wrong but could not pinpoint the error specifically. Consider this APROPOS1's analysis of a student's reverse/2 program :

Your program clauses are :

```
reverse([], []).
reverse([Head1|Tail1], [Something|Head1]) :-
    reverse(Tail1, Something).
```

Warning. In the clause

```
reverse([Head1|Tail1], [Something|Head1]) :-
    reverse(Tail1, Something).
```

if Head1 in [Something|Head1] is not a list, perhaps, an atom, then [Something|Head1] may not be what you want.

A note about syntax : [a|[b]] is equivalent to [a,b], and [a|[b,c]] is equivalent to [a,b,c] but [a|b] is not the same as [a,b].

Your base clause of reverse/2 :

```
reverse([], []).
```

seems fine.

Let's run your program on an example : reverse([], X).

Your program produces solution : reverse([], []) which is correct, and no more solutions.

Analysing clause :

```
reverse([Head1|Tail1], [Something|Head1]) :-
    reverse(Tail1, Something).
```

Fail to understand why you have [Something|Head1] in the head goal and Something in the recursive goal.

Let's run your program on an example : reverse([a,b], X).

Your program produces solution : reverse([a,b], [[[]|b]|a]) which is incorrect.

Error : wrong solution reverse([a,b], [[[]|b]|a]). Diagnosing ...

Error diagnosed :

```
reverse([b]|[[[]|b]]) :- reverse([], []).
```

is the source of the error. The goal(s) in reverse([], []) is correct but reverse([b], [[[]|b]] is not true.

The clause at fault may be :

```
reverse([Head1|Tail1], [Something|Head1]) :-
    reverse(Tail1, Something).
```

{ the session continues }

Figure 6-4 : APROPOS1's inability to suggest correction to a buggy clause

As algorithm recognition is by feature identification, APROPOS1 failed to match the student's program to one of the algorithms it knew of, since it could not identify enough algorithmic features. Thus it could not suggest what was the bug or bugs in the buggy recursive clause. With no algorithm matched, there was also no context of what clause to compare with. Consequently, it could not comment on the program. APROPOS1 falls back on using Shapiro's diagnosis routines. We had intended APROPOS1 to only show the source of the error in order to present only precise information to the student. One student who wrote this version of the program found this help provided by APROPOS1 to be useful. Yet another student who wrote the same version and therefore got the same APROPOS1's output commented "*the explanation of the wrong solution and the following diagnosis felt like a sugar-coated traceback from the point of failure. An additional explanation of the failure in terms of the wrong goal being generated right at the beginning may have been useful in getting*

me to think in the right direction." This is what Shapiro's routines actually avoid as it presents only the source of the error instead of the whole trace. As we have not begun to incorporate student modelling in it, APROPOS1 uses a *first-order theory* of advice-giving – it produces the same output for the same bug encountered. However as no one explanation suits every student, it would be nice but difficult to customise explanations for each student.

In the cases where APROPOS1 could pinpoint the error, other students commented that while it did tell them what was wrong with their program, it did not explain well why their code was incorrect or inefficient. Thus the advice given was not useful to them as to how to correct their program. Consider this APROPOS1's analysis of a student's reverse/2 program :

Your program clauses are :

```
reverse([], []).
reverse([X|Rest], Result) :-
    append(Start, [X], Result),
    reverse(Rest, Start).
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :-
    append(Xs, Ys, Zs).
```

Your base clause of reverse/2 :
reverse([], []).

seems fine.

Let's run your program on an example : reverse([], X).

Your program produces solution : reverse([], []) which is correct, and no more solutions.

Analysing clause :

```
reverse([X|Rest], Result) :-
    append(Start, [X], Result),
    reverse(Rest, Start).
```

You have got the declarative logic right, but append(Start, [X], Result) should for efficiency sake come *after* the recursive goal. Otherwise, this clause looks fine. It has a recursive goal and a clause to perform on Start to get Result.

{ the session continues }

Figure 6-5 : APROPOS1's too terse explanation capability

The student commented "*With my second try, it correctly pointed out that I should have the goals in the body in a different order, stating that this was so for efficiency reasons. However, it didn't state why this version was inefficient, nor did it tell me that this version would go into infinite regress if an attempt was made to resatisfy the goal after a successful reverse. (I found out this fact by trial and error). ... It correctly diagnosed a fault. However, it was little use in explaining why it was a fault and how inefficient my program would be. I would have appreciated either a flow-of-control justification for its comment, and/or some statement (maybe just sample run-times) of how inefficient my program would become.*" We can meet part of this criticism by adding a more intricate explanation, which may include a discussion

of a run-time trace, to the template commentary associated with this common bug. This also provide further motivation to work on a more flexible analysis of the run-time behaviour of Prolog programs, especially using modes. We will discuss this further in Section 8.4.

As our emphasis has been on the bug diagnosis capability of APROPOS1, we have not taken painstaking care to phrase the help messages for each type of bug. Nevertheless, making APROPOS1's critiques in the form of template commentaries more decorous will help make its advice messages more understandable to a wider range of students.

In summary, running the set of collected programs on APROPOS1 as well as analysis of students' interactive sessions with APROPOS1 suggest that APROPOS1's approach is feasible for the diagnosis of errors in actual students' programs. However, its performance depends on its ability to detect sufficient features from the program to aid its analysis. If it cannot detect enough features, APROPOS1 will not be able to comment usefully on some parts of the student's programs.

APROPOS1 can only be used to analyse a small set of recursive tasks, like `reverse/2`, `length/2` and `factorial/2`, which have a common recursive structure. To improve its applicability, we adopted a heuristic code-matching approach which matches students' programs against reference P-frames. This has been incorporated in APROPOS2.

While some students found APROPOS1's advice messages to be helpful for correcting their buggy programs, and gave positive comments on it, other students needed more help than is currently available with these messages. That some students were able to arrive at a first correct version of `reverse/2` after an interactive version with APROPOS1 which had criticised their buggy programs attests some educational value in using APROPOS1. However, we state no claim about the educational effectiveness of using APROPOS1 nor have we set out to evaluate its pedagogical value as the intention of developing APROPOS1 is exploratory.

6.4 Analysis-by-Synthesis in APROPOS-AS

A second system APROPOS-AS was built to explore an analysis-by-synthesis approach to analyse failure-driven loops in Prolog. Debugging failure-driven loops requires a more involved analysis of feature extraction because there can be many different ways of implementing failure-driven loops, and a dataflow analysis is at least required. As programming tasks (whose realisation involves failure-driven loops) have task-specific goals, we want to explore the use of an analysis-by-synthesis approach to analyse how task goals may have been implemented in the student's program. Like PROUST, we want to explore an interpretation space where each state consists of the following :

- an agenda of goals whose implementation in the program has yet to be determined,
- plans which have been matched against the program, and the code they match.

The simple programming task which we have set APROPOS-AS to analyse is :

Task : CALCULATING POPULATION-DENSITY

Asserted in the Prolog database are clauses :

% figures given in millions

pop(britain, 56).

pop(india, 685).

... more data ...

% figures given in thousands of square miles

area(britain, 94).

area(india, 1262).

... more data ...

Write a Prolog program density/0 which, for each country C, finds its population P and its area A, and calculates its population density D, and prints C, P, A, D.

Two goals for this problem are to compute the density and to print it out. It is only appropriate to comment on the second goal (that it is not met, for example) only if the first goal is satisfied. To illustrate why we should reformulate task goals, consider this program :

```
density :-
    do_the_work.

do_the_work :-
    do_once_only,
    pop(C, Pop),
    area(C, Area),
    Density is Pop/Area,
    { write subgoals },
    fail.
```

Such a predicate like do_once_only/0 might be used for printing beautiful titles. One goal in the problem description is to have a clause *to make the call to density/0 succeed* when all the

work is done. For this particular program, there are two ways of doing this : to add an unit clause :

density.

or

do_the_work.

However, if there is a cut which appears immediately after the `do_once_only` subgoal in the `do_the_work` clause, then the first way is the right one to write a catch-all clause to make the loop call succeed. Thus the task goal to have an unit clause to make the loop call succeed needs to be dynamically elaborated as the analysis proceeds to find if there is a cut in the clause for `do_the_work`. Here the task goal is elaborated by adding a constraint that it is to be realised only in the `density/0` predicate definition instead of the `do_the_work/0` predicate definition.

Subsequent to our initial effort in building a Prolog analyser APROPOS1 which debugs simple recursive programs, we built APROPOS-AS to explore an analysis-by-synthesis approach for analysing Prolog programs that handle multiple goals, such as for the above problem.

The initial representation of the population density problem in APROPOS-AS's problem description language is a `'&agenda'/7` predicate which is read as follows :

```
'&agenda'( goal name,  
            plan template,  
            predicate to realise goal to code,  
            is goal realised?,  
            is goal activated?,  
            where is goal realised in code?,  
            description of goal ).
```

Essentially, each of the predicates mentioned in argument 3 of `'&agenda'/7` tries to realise a goal in different alternative plans, that is, each clause of the predicate realises or synthesizes the program code as a different plan. There can be much variability in the code students use to realise goals. To facilitate analysis, a plan like :

```
loop(...) :-  
    loop1(...).  
loop(...).  
  
loop1(...) :-  
    do-once,  
    !,  
    ...,  
    fail.
```

is distinguished from another plan like :

```
loop(...) :-  
    do-once,  
    fail.  
loop(...) :-  
    loop1(...).  
  
loop1(...) :-  
    ...,  
    fail.  
loop1(...).
```

That there can be many ways of implementing a failure-driven loop in Prolog suggests the need to represent several plans which can meet this goal. Thus we represent common variants of a loop as plans, such as these :

PLAN 1 :

```
loop(...) :-  
    foo(...).  
loop(...).           % catchall clause to make a call to loop succeed  
foo(...) :-  
    once,             % subgoals intended to be invoked once only  
    !,  
    db(X),            % fetch data from data-base  
    write(X),         % side-effecting subgoals to handle the data  
    fail.
```

PLAN 2 :

```
loop(...) :-  
    foo(...).  
foo(...) :-  
    once,             % subgoals intended to be invoked once only  
    db(X),            % fetch data from data-base  
    write(X),         % side-effect subgoals to handle the data  
    fail.  
foo(...).           % catchall clause to make a call to loop succeed
```

PLAN 3 :

```
loop(...) :-  
    foo(...).  
foo(...) :-  
    fie(...),  
foo(...).           % catchall clause to make a call to loop succeed  
fie(...) :-  
    once,             % subgoals intended to be invoked once only  
    !,  
    db(X),            % fetch data from data-base  
    write(X),         % side-effect subgoals to handle the data  
    fail.
```

The goals for the population density problem are :

```
'&agenda'(loop-plan, -, predict-loop, unrealised, activated, unlocated, '... a loop to retrieve all
data in database ...').
'&agenda'(succeed-eventually, -, predict-catchall, unrealised, activated, unlocated, '... to have
a clause that makes the failure-driven loop succeed eventually ...').
'&agenda'(fetch-pop, [pop, C, P], predict-resatisfiable-goal, unrealised, activated, unlocated,
'... fetch the population of a country ...').
'&agenda'(fetch-area, [area, C, A], predict-resatisfiable-goal, unrealised, activated,
unlocated, '... fetch the area of a country ...').
'&agenda'(compute-density, [' ', P, A], predict-compute-goal, unrealised, activated, unlocated,
'... compute the density of a country ...').
'&agenda'(output-country, -, predict-side-effect-goal, unrealised, unactivated, unlocated, '...
print the country name ...').
'&agenda'(output-pop, -, predict-side-effect-goal, unrealised, unactivated, unlocated, '... print
the population figure ...').
'&agenda'(output-area, -, predict-side-effect-goal, unrealised, unactivated, unlocated, '...
print the country's area ...').
'&agenda'(output-density, -, predict-side-effect-goal, unrealised, unactivated, unlocated, '...
print the density ...').
```

APROPOS-AS puts activated goals onto the goal agenda for further analysis. As in PROUST, we order goal analysis by selecting first those goals that define the larger control structures in the program, as in the goal *loop-plan* for the population-density problem. As the goal for computing population density depends on the goals to fetch the population and area, we order the latter goals first. There are also other goal dependencies information, such as, that the goal for *output-pop* is only activated only when the goal for *fetch-pop* has been found to be accomplished. As described before, goals can be elaborated. For example, in trying to match this code to the goal *loop-plan*,

```
do_the_work :-
    do_once_only,
    !,
    pop(C, Pop),
    area(C, Area),
    Density is Pop/Area,
    { write subgoals },
    fail.
```

the *succeed-eventually* goal is elaborated to :

```
'&agenda'(succeed-eventually, List-of-Calling-Predicates, predict-catchall, unrealised,
activated, unlocated, '... to have a clause that makes a call to the failure-driven
loop succeed eventually ...').
```

where List-of-Calling-Predicates is the list of parent predicates that call *do_the_work/0*. To satisfy this goal, the clause to make the failure-driven loop succeed must eventually come from the List-of-Calling-Predicates.

APROPOS-AS explores a *state space* where each state consists of the status of the goals (both activated and unactivated goals), and the plans which have been matched against the

program, and the code they match. In the initial state, the goal agenda is the set of activated goals in the problem description. In each possible final state, the agenda has all activated goals realised, meaning that all goals in the problem description, as well as all elaborated goals, have been accounted for.

How might one deal with the problem of alternative interpretations of the code which are likely to occur if there are multiple bugs that complicate the analysis? PROUST's approach is to decide early on one best interpretation that has the least mismatches between goal and code, and to repair it later if necessary. We propose to keep a number of possible interpretations all active simultaneously and to use more information to decide which interpretation to proceed on next. A suitable approach for doing this is a truth maintenance system [Doyle 79].

6.5 Empirical Evaluation of APROPOS-AS

To find out how well APROPOS-AS can analyse actual student's programs for the POPULATION-DENSITY task, we asked 5 volunteers from the AI-2 class of 1986/87 and 5 volunteers from the MSc class of 1986/87 to try it interactively. Three of the 5 AI-2 students and three of the MSc students wrote correct programs on their first attempts. The other students started off with incorrect programs but were able to arrive at a correct solution of the program. APROPOS-AS produced useful analyses for 7 of these 10 students. The programs written by these students used plans which APROPOS-AS could analyse. APROPOS-AS failed in its analysis of three unusual but correct MSc students' programs.

Appendix III shows a session listing of APROPOS-AS's analyses of a student's programs for the POPULATION-DENSITY task. Figure 6-5 shows APROPOS-AS's analysis of a student's solution :

Your program clauses are :

```
writeout(C, P, A, D) :-
    write(C),
    write(' '),
    write(P),
    write(' '),
    write(A),
    write(' '),
    write(D).

density :-
    write(' Country Population(m) Area(m sq m) Density'),
    !.
density :-
    pop(C, P),
    area(C, A),
    D is P/A,
    writeout(C, P, A, D).
```

Now, this is APROPOS-AS's critique on your program :

This failure-driven loop you intend to have in

```
density :-
    pop(C, P),
    area(C, A),
    D is P/A,
    writeout(C, P, A, D).
may not be invoked at all!
```

This clause :

```
density :-
    write(' Country Population(m) Area(m sq m) Density'),
    !.
may succeed because it calls no fail goal.
```

This clause :

```
density :-
    pop(C, P),
    area(C, A),
    D is P/A,
    writeout(C, P, A, D).
```

does not process all the data because there is no loop. You need to use a loop and in Prolog, a loop can be implemented using the fail goal to backtrack to retrieve all the data eventually. This is called a failure-driven loop.

Warning : the cut removes all backtracking points in the invocation of the predicate.

>>> One of the goals for this task is a loop to retrieve all data in the database. You do not seem to have achieved this goal.

Figure 6-6 : An example of APROPOS-AS's analysis

APROPOS-AS traces the dataflow of the C, P, A and D terms from the density/0 predicate definition to the write goals. Such a dataflow analysis is necessary, for there can be many implementation variants whose analyses require a dataflow trace of the terms C, P, A and D

in the main density/0 predicate definition. To illustrate, these can be two of the many possible ways :

```
density :-
    fetch1(C, P, A),
    write(C),
    ...
fetch1(C1, P1, A1) :-
    fetch(A1, P1, C1),
    ...
fetch(A, P, C) :-
    pop(C, P),
    area(C, A),
    ...
```

or

```
density :-
    fetch1(C, P, A, D),
    write(C),
    ...
fetch1(C1, P1, A1, D) :-
    fetch(A1, P1, C1, D),
    ...
fetch(A, P, C, D) :-
    pop(C, P),
    area(C, A),
    ...
```

It seems unlikely that students will write such wayward variants, but a robust program analyser should at least be flexible enough to do a dataflow analysis that can understand different possible implementations of code such as these.

APROPOS-AS's analysis fails when it comes to analysing programs of three of the students which use quite unusual ways :

```
PROGRAM 1 :
d([C,P,A,D]) :-
    pop(C, P),
    area(C, A),
    D is P/A.
density :-
    setof(X, d(X), S),
    writeS(S).
writeS([]) :-
    nl.
writeS([H]) :-
    !,
    write(H),
    nl.
writeS([H|T]) :-
    write(H),
    write(' '),
    writeS(T).
```

PROGRAM 2 :

```
density :-
    write(' Country Population(m) Area (m sq m) Density'),
    nl,
    bagof(Country, do_data(Country), Set).
do_data(Country) :-
    pop(Country, X),
    area(Country, Y),
    write(Country),
    tab(10),
    write(X),
    tab(10),
    write(Y),
    Z is X/Y,
    tab(10),
    write(Z).
```

PROGRAM 3 :

```
density :-
    nl,
    write('Country'),
    tab(7),
    write('Pop (m)'),
    tab(3),
    write('Area'),
    tab(5),
    write('Density'),
    \+ countries_write.
countries_write :-
    pop(C, P),
    area(C, A),
    D is P/A,
    write(C),
    tab(10),
    write(P),
    tab(5),
    write(A),
    tab(5),
    write(D),
    nl,
    fail.
```

This illustrates the limitation of matching a student's program against a library of plans. Using the different primitives of the Prolog language, there can be many plans to realise a goal. Programs 1 and 2 make use of the `setof/3` and `bagof/3` predicates in Prolog, while Program 3 makes use of `\+`, all of which APROPOS-AS does not do any computational reasoning about. The programs work for the task they are set to do, that is, compute the population densities of the countries in the database. While it is difficult for an automated debugger to understand these programs, it must also be said a hand analysis of whether they work or not is not an easy task to do.

This highlights one difficulty of an analysis-by-synthesis approach for debugging Prolog programs : it is difficult or impossible to enumerate all the different ways a plan can be implemented in Prolog which includes implementations with convoluted control flow that make use of the semantics of Prolog primitives and specific features of Prolog execution. One

implication is that the analysis-by-synthesis approach in APROPOS2 has inherent limitations in analysing Prolog programs.

6.6 Summary

In summary, we have explored an analysis-by-synthesis approach to program analysis in Prolog. We built APROPOS-AS which diagnoses bugs in failure-driven loops for a specific task by synthesizing different implementations of loops and matching these implementations to the student code. The approach is reasonable but limited as it depends on having a library of implementation plans representative of the kind of code students typically write. We have also used an analysis-by-synthesis approach in APROPOS2 but in a different way. APROPOS2 analyses a student predicate definition for a task by selecting an algorithm for the task, synthesizing the different implementations of the algorithm and searching which best matches the student program. This approach is more suitable for recursive list and number manipulation programs which usually use auxiliary predicates whose implementations can be analysed in a similar way using the same approach recursively.

The intention of developing APROPOS1 and APROPOS-AS is very exploratory. The process of building and testing these programs gives us confidence in the utility of combining dynamic with static analysis, and of an analysis-by-synthesis approach. At the same time, we want to improve the applicability of these analysis approaches to handle more tasks and to separate the task knowledge from the code analysis process. With these considerations in mind, we have developed APROPOS2 which uses a best-first search to do code-matching. Our empirical experiments with APROPOS1 also allowed us to collect a set of students' reverse/2 programs which are later used to test APROPOS2.

Chapter 7. Empirical Evaluation of APROPOS2

We have described the debugging approach used in APROPOS2 which is intended to be used as the expertise module in a PITS. The bottom line of an ITS is its actual performance on real tasks. In this chapter, we discuss our empirical evaluation of APROPOS2's performance in its task of debugging various solutions for simple but nontrivial tasks.

7.1 Set-Up of the Evaluation

In Chapter 7, we described an initial version of APROPOS (APROPOS1). APROPOS1 does no best-first search. Instead, it extracts features such as dataflow in the iteration and recursion arguments, and programming techniques such as the use of an accumulator argument from the code to recognise the algorithm used. Some of these feature extractions are specific to the reverse/2 task. The current version of APROPOS (APROPOS2) is an attempt to generalise the debugging algorithm so that it can be used for a wider range of tasks. As described in this thesis, APROPOS2 incorporates a best-first search to interpret the student program. As the code analysis process is independent of the task examined, it can be used to analyse a new task by encoding the task knowledge. Currently, the tasks that have been encoded are :

- reverse/2 to reverse a list
- sorting/2 to sort a list
- replace_elements/4 to replace all occurrences of one element in a list by another
- atom_count/2 to count the number of atoms in a list
- fringe_length/2 to count the number of atoms in all levels of a list

We evaluated APROPOS2 on the corpus of students' reverse/2 programs collected in the empirical trials of APROPOS1. APROPOS2 has also been put to the test on three of many programming tasks given to the MSc Prolog class of 1986/87 in the Department of AI, University of Edinburgh. These tasks concern list or number manipulation. There are 22 programs for each of the tasks collected by the tutors. There was a high percentage of correct programs as these were submitted programs. We also report on two student volunteers who tried the sorting/2 task interactively on APROPOS2.

In the next sections of this chapter, we discuss empirical results of running APROPOS2 on this corpus of students' programs. To evaluate its performance, we need to judge how well has APROPOS2 recognised algorithms, detected bugs and suggested corrections in the students'

programs. We argue that the important part is algorithm recognition, because with correct recognition of the algorithm, the bug detection and correction will follow naturally from APROPOS2's code-matching and code-critique phases. APROPOS2's bug critique is based on a model of an ideal reference program and its code-matching suggests the best fit with the least discrepancies according to APROPOS2's measure of code mismatch. In this sense bug corrections are minimal fixes to the program. In what follows, the judgement of whether APROPOS2 has detected an algorithm correctly for a given program is the author's. To seek a fair and independent evaluation, we gave the same corpus of programs to a Prolog lecturer in the department and asked him to detect the algorithms attempted in the programs and the bugs in the incorrect programs. Section 7.7 discusses our observations on this independent evaluation.

We judge a solution for a task to be *correct* if it uses some algorithm for doing the task and the first solution it produces is the correct answer. Judging that the first solution returned is the correct answer seems a reasonable and not too strict test for novice programs. This seems what novices have to be preoccupied with, getting the solution at all in the first place. If we are to judge a program for a determinate task like *reverse/2* or *sorting/2* to be correct only if it produces one and only one solution on the initial call and on forced backtracking, then very few of the students' programs will meet this criterion. We also consider programs which run inefficient algorithms but produce the correct answer on first success to be correct even though APROPOS2's message templates do incorporate some but not much commentaries on the inefficiency of algorithms.

APROPOS2 judges a program to be correct from its heuristic code-matching process, that is, if an algorithm has been matched, the reference predicates matched are correct solutions, and there is no discrepancy in the match which cannot be accounted for. This is the static test for program correctness. The interactive version of APROPOS2 runs the student solution on some test cases, and collects all the solutions produced. This is the dynamic test for program correctness. We can therefore adopt a better criterion for program correctness by judging a program to be correct only if it meets both these two tests of code and dynamic analysis. As we want to evaluate APROPOS2's static code-matching approach, we will judge its static test for program correctness.

Our definitions of the different bug types have been described in Section 3.2. An instance of each of these is counted as a bug :

- missing clause
- extra clause
- missing subgoal
- extra subgoal
- wrong subgoal

- missing argument position
- wrong subgoal order
- wrong clause order

The figures for algorithm recognition are obtained only for programs for which APROPOS2 has a complete analysis. APROPOS2 does an incomplete analysis if it is not able to find an algorithm match for the main student predicate definition. This happens when the best algorithm match it finds exceeds the threshold for an acceptable match. The figures for bug detection and correction are obtained only from the programs whose algorithms are correctly recognised by APROPOS2 since bug detection and correction make sense only when algorithms are correctly recognised. This will also allow us to isolate the strengths and weaknesses of APROPOS2.

7.2 The reverse/2 task

As described in Section 4.3, we use an initial set of 40 hand-written solutions to help set up APROPOS2's reference frames. We ran APROPOS2 on all our collected reverse/2 programs. We considered only syntactically correct programs and collect all the different versions from interactive sessions. We collected 95 programs of which there were 55 different variations if we ignore a relabelling of variables and inconsequential clause orderings. Of these 95 programs :

- 49 attempted the naive method,
- 16 attempted the railway-shunt method using an accumulator,
- 1 attempted the inverse naive method,
- 23 attempted code which has a clause similar to

```
reverse([H|T], Y) :-
    reverse(T, [H|Y]).
```

This suggests that they may be trying the railway-shunt method and misconceive that the final answer can somehow be obtained.

- 6 attempted code for which it is unclear what algorithm has been intended.

The results of the analyses of these 95 reverse/2 programs are :

Total number of students' programs for reverse/2 :	95
No. of correct programs	26
No. of correct programs judged as correct by APROPOS2	24
No. of correct programs judged as incorrect by APROPOS2	2
No. of incomplete analyses by APROPOS2	0
No. of incorrect programs	69
No. of incorrect programs judged as correct by APROPOS2	1
No. of incorrect programs judged as incorrect by APROPOS2	68
No. of incomplete analyses by APROPOS2	0

Table 7-1 : Results of APROPOS2's analyses of correctness of students' reverse/2 programs

Algorithm Recognition :

No. of programs whose algorithms are correctly identified by APROPOS2	76
No. of correct programs	26
No. of correct programs judged as correct by APROPOS2	24
No. of correct programs judged as incorrect by APROPOS2	2
No. of incorrect programs	56
No. of incorrect programs judged as correct by APROPOS2	1
No. of incorrect programs judged as incorrect by APROPOS2	55
No. of programs whose algorithms are incorrectly identified by APROPOS2	13
No. of correct programs	0
No. of correct programs judged as correct by APROPOS2	0
No. of correct programs judged as incorrect by APROPOS2	0
No. of incorrect programs	13
No. of incorrect programs judged as correct by APROPOS2	0
No. of incorrect programs judged as incorrect by APROPOS2	13
No. of programs whose algorithms are unclear	6

Table 7-2 : Results of APROPOS2's analyses of algorithm recognition of students' reverse/2 programs

Bug Detection and Correction :

Total no. of bugs	115
Total no. of bugs detected and corrected	114
Total no. of bugs detected but corrected wrongly	0
Total no. of bugs not detected	1
Total no. of false alarms	5

Table 7-2 : Results of APROPOS2's analyses of bug detection and correction of students' reverse/2 programs

Appendix III lists APROPOS2's analyses of four students' programs which illustrate the range of programs APROPOS2 analyses. In this section, we discuss a number of program analyses which bring features of APROPOS2's debugging approach into focus, and demonstrate APROPOS2's capabilities as well as its limitations.

Figure 7-1 shows the analysis of a program for which APROPOS2 has two algorithm interpretations. APROPOS2 finds algorithm matches from both the naive reverse/2 method and the inverse naive reverse/2 method (which takes out the last element of the list, reverses all the other elements and splice the last element back) and suggests the bug corrections, even though it seems that the student has attempted the naive method of reverse/2 instead of the inverse naive method. APROPOS2 provides the missing predicate definition of add/3. This demonstrates that APROPOS2 is able to fill in the missing part of the student's program, not that it should. Whether a PITS incorporating APROPOS2 should ask the student for the missing code or provide the missing code itself is a matter involving student modelling and tutoring strategy. For example, it may be useful for the student modeller to know whether it is bad planning, or forgetfulness or a misconception that add/3 is a system predicate, that causes the student to fail to provide the missing code. Some students do think append/3 is a system predicate.

Another way of fixing the student program under the first program interpretation is to leave the add/3 subgoal as it is, and include the appropriate add/3 definition :

```
add([H|X], Y, [H|Z]) :-
    add(X, Y, Z).
add([], L, [L]).
```

APROPOS2 does not provide this fix as we did not want to incorporate this much less applicable implementation of append/3 into the append/3 reference P-frame.

Your program clauses are :

```
reverse([], []).
reverse([H|T], X) :-
    add(H, X, Y),
    reverse(T, Y).
```

Your program seems to be incorrect.
Now, this is APROPOS2's critique on your program :

*** Program Interpretation No 1 ***

Clause 1 of reverse/2 seems fine.
The reverse of the empty list is the empty list.

Clause 2 of reverse/2 does not seem quite right.
Replace add(H, X, Y) with add(Y, [H], X).
This recursive case says that the reverse of a non-empty list can be found by reversing the tail of the list, and then appending a list consisting only of the first element of the original to the result of reversing the tail. However, you should put the add subgoal after the recursive subgoal for efficiency reasons. If it comes before the recursive subgoal, then Y and X will be uninstantiated variables when this goal is attempted. It will try many values for Y and X before it gets the "right" one.

This predicate definition seems to be missing :

```
add([H|X], Y, [H|Z]) :-
    add(X, Y, Z).
add([], L, L).
```

The first clause is the recursive clause.
The second clause says that the result of appending an empty list with the list L is the list L itself.

*** End of Program Interpretation No 1 ***

*** Program Interpretation No 2 ***

Clause 1 of reverse/2 seems fine.
The reverse of the empty list is the empty list.

Clause 2 of reverse/2 does not seem quite right.
Replace reverse([H|T], X) with reverse(H, [X|Y]) and
add(H, X, Y) with add(H, X, T).

This recursive case says that the reverse of a non-empty list can be found by reversing the list without the last element, and then appending a list consisting only of the last element of the original with the result of reversing the list without the last element.

This predicate definition seems to be missing :

```
add([Last], Last, []).
add([A|List], Last, [A|Rest]) :-
    add(List, Last, Rest).
```

Given a list as input, this predicate definition gets the last element of the list as well as a list containing all the elements of the original list without the last element.

*** End of Program Interpretation No 2 ***

Figure 7-1 : Two best algorithm matches for a reverse/2 program by APROPOS2

Figure 7-2 shows the analysis of a correct program which was judged as incorrect by APROPOS2. This student program is an implementation of the railway-shunting reverse using an accumulator. But as it disguises the result variable in the second argument of `rev/2`, APROPOS2 did not accept it as correct from its heuristic code-matching and code-critique process since it does not have a representation of this two-argument version of the railway-shunting reverse/2. A dynamic analysis on three test data shows that the student's program produces correct results on the first run.

This analysis illustrates the limitation of APROPOS2's debugging approach in that it can fail in its analysis of programs that use novel implementations of algorithms. APROPOS2 can fail to recognise the intended algorithm used in the student's predicate definition if the predicate uses a different number of arguments. A more canonical representation of program dataflow may be able to match different implementations of the same dataflow but this poses the difficulty of providing useful contextual advice.

Your program clauses are :

```
reverse(A, B) :-
    rev(A, [B]).
rev([], [B|B]).
rev([Head|Tail], [B|Temp]) :-
    rev(Tail, [B,Head|Temp]).
```

Your program seems to be incorrect.
Now, this is APROPOS2's critique on your program :

Clause 1 of `rev/2` does not seem quite right.
This is what your clause should look like :
 `rev([], X, X).`
This clause instantiates the accumulated result to the result variable.

Clause 2 of `rev/2` does not seem quite right.
This is what your clause should look like :
 `rev([Head|Tail], X, Y) :-`
 `rev(Tail, [Head|X], Y).`
This clause uses an argument to accumulate the result as recursion proceeds, and another argument to be instantiated to the accumulated result when recursion ends.

Clause 1 of `reverse/2` does not seem quite right.
This is what your clause should look like :
 `reverse(A, B) :-`
 `rev(A, [], B).`
`reverse/2` calls another predicate to do the work.

Figure 7-2 : A correct `reverse/2` program judged as incorrect by APROPOS2

Figure 7-3 shows the analysis of a program which is interesting as it has several bugs in it, some of which are more serious than the others. APROPOS2's analysis shows that it can recognise the algorithm used. However, APROPOS2 fails to match the first clause of the

student's reverse/3 with the base clause of the railway-shunt reverse. Five bugs are counted in this buggy program, as shown in the italic annotations in the figure. This way of counting of bugs can be improved since it disguises certain information. We had chosen this way of counting bugs as it falls naturally into APROPOS2's scheme of detecting bugs and suggesting bug corrections at the literal level. One observation from studying this student's program at closer detail is that the main bug of this program is in the third argument of the student's reverse/3. The bug arising from the use of atom(List) and \+ atom(List) is not so serious since atom([]) succeeds, and so apart from the main bug would reverse non-empty lists. Nevertheless, this kind of analysis requires at least a more intricate analysis than is currently available in APROPOS2. As it is, APROPOS2 describes the bugs at the literal level but it may be better, at least sometimes, to describe the bug at the argument level across the same predicate at different places of the program, as for this student's program. Analysing the severity of different bugs in a program and deciding which should be the main bug(s) to address first is a matter for further study.

Your program clauses are :

```
reverse(List, Tsil) :-
    reverse(List, [], Tsil).
reverse([X|List], Temp, [X|Temp]) :-
    atom(List).
reverse([X|List], Temp, Temp) :-
    \+ atom(List),
    reverse(List, [X|Temp], [X|Temp]).
```

Your program seems to be incorrect.
Now, this is APROPOS2's critique on your program :

Clause 1 of reverse/3 seems to be redundant. *{bug}*

Clause 2 of reverse/3 does not seem quite right.

Replace reverse([X|List], Temp, Temp) with reverse([X|List], Temp, Y) *{bug}*
and reverse(List, [X|Temp], [X|Temp]) with reverse(List, [X|Temp], Y). *{bug}*
\+ atom(List) seems to be redundant. *{bug}*

This is what your clause should look like :

```
reverse([X|List], Temp, Y) :-
    reverse(List, [X|Temp], Y).
```

This clause uses an argument to accumulate the result as recursion proceeds, and another argument to be instantiated to the accumulated result when recursion stops.

In your predicate definition of reverse/3, a clause like :

```
reverse([], X, X).
```

seems to be missing. *{bug}*

This clause instantiates the accumulated result to the result variable.

Clause 1 of reverse/2 seems fine.

reverse/2 calls another auxiliary predicate reverse/3 to do the job.

Figure 7-3 : A correct algorithm match of a reverse/2 program by APROPOS2

Figure 7-4 shows the analysis of the other correct program which was judged as incorrect by APROPOS2. The program is considered to be correct as it produces the answer on the first successful goal execution, although the flow of control is rather convoluted. One way for APROPOS2 to be able to accept this program as correct from its code-matching process is to add knowledge about symmetry of relations into the task knowledge that APROPOS2 uses. Thus reverse/2 is symmetric by its definition and the heuristic-code matching process would accept as an equivalent match an instance of reverse(Arg1, Arg2) with reverse(Arg2, Arg1). If arguments are swapped in this way, the run-time behaviour may give undesirable results. We would need a further analysis with modes and dynamic analysis to help predict the run-time behaviour. This will be discussed further in Section 8.4.

Your program clauses are :

```
reverse([], []).
reverse([Head|Tail], X) :-
    append(Y, [Head], X),
    reverse(Y, Tail).
append([], List, List).
append([H|T], List2, Result) :-
    Result = [H|X],
    append(T, List2, X).
```

Your program seems to be incorrect.

Now, this is APROPOS2's critique on your program :

Your clause

```
append([H|T], List2, Result) :-
    Result = [H|X],
    append(T, List2, X).
```

can be rewritten as :

```
append([H|T], List2, [H|X]) :-
    append(T, List2, X).
```

Clause 1 of append/3 seems fine.

The first clause is the recursive clause.

Clause 2 of append/3 seems fine.

The second clause says that the result of appending an empty list with the list L is the list L itself.

Clause 1 of reverse/2 seems fine.

The reverse of the empty list is the empty list.

Clause 2 of reverse/2 does not seem quite right.

Replace reverse(Y, Tail) with reverse(Tail, Y).

This recursive case says that the reverse of a non-empty list can be found by reversing the tail of the list, and then appending a list consisting only of the first element of the original to the result of reversing the tail. However, you should put the append subgoal after the recursive subgoal for efficiency reasons. If it comes before the recursive subgoal, then Y and X will be uninstantiated variables when this goal is attempted. It will try many values for Y and X before it gets the "right" one.

Figure 7-4 : Another correct reverse/2 program judged as incorrect by APROPOS2

Figure 7-5 shows the analysis of the incorrect program which was judged as correct by APROPOS2. The program is incorrect, as the `app1` subgoal will be invoked with a mode of `app1(-, +, -)` which loops since the `app1/3` recursive clause is placed before the base clause. APROPOS2 accepts the program as correct as no clause ordering constraint on the `append/3` predicate definition was included in the inefficient naive program representation in the reference P-frame for naive reverse and it did not check for this constraint. Adding this clause ordering constraint will correct APROPOS2's analysis. Our intention is to alleviate the use of a slot for clause ordering constraint in a reference P-frame and instead use mode analysis to help reason about clause orderings. A dynamic analysis in the interactive version of APROPOS2 will tell that `reverse(+, -)` will result in nontermination.

Your program clauses are :

```
app1([H|L1], L2, [H|L3]) :-
    app1(L1, L2, L3).
app1([], L, L).
reverse([], []).
reverse([H|T], Res) :-
    app1(Sofar, [H], Res),
    reverse(T, Tail).
```

Your program seems to be correct.

Now, this is APROPOS2's critique on your program :

Clause 1 of `append/3` seems fine.

The first clause is the recursive clause.

Clause 2 of `append/3` seems fine.

The second clause says that the result of appending an empty list with the list `L` is the list `L` itself.

Clause 1 of `reverse/2` seems fine.

The reverse of the empty list is the empty list.

Clause 2 of `reverse/2` seems fine.

This recursive case says that the reverse of a non-empty list can be found by reversing the tail of the list, and then appending a list consisting only of the first element of the original to the result of reversing the tail. However, you should put the `app1` subgoal after the recursive subgoal for efficiency reasons. If it comes before the recursive subgoal, then `Sofar` and `Res` will be uninstantiated variables when this goal is attempted. It will try many values for `Sofar` and `Res` before it gets the "right" one.

Figure 7-5 : An incorrect `reverse/2` program judged as correct by APROPOS2

All 13 programs whose algorithms are incorrectly identified by APROPOS2 have a clause similar to

```
reverse([H|T], Y) :-  
    reverse(T, [H|Y]).
```

It would seem the intended algorithm in these programs is the railway-shunt method but as these reverse/2 programs have only two argument positions, APROPOS2's code-matching suggests for most of these programs the naive method as the best-fit. It may be possible to improve APROPOS2's performance in this respect by giving less weight to the comparison of the number of arguments.

Figure 7-6 shows the analysis of a program whose algorithm seems to be incorrectly judged by APROPOS2 since it seems that the student was attempting the railway-shunt method of reverse.

Your program clauses are :

```
reverse([], []).  
reverse([H|T], Y) :-  
    reverse(T, [H|Y]).
```

Your program seems to be incorrect.

Now, this is APROPOS2's critique on your program :

Clause 1 of reverse/2 seems fine.

The reverse of the empty list is the empty list.

Clause 2 of reverse/2 does not seem quite right.

Replace reverse(T, [H|Y]) with reverse(T, X)

and append(X, [H], Y) seems to be missing.

This is what your clause should look like :

```
reverse([H|T], Y) :-  
    reverse(T, X),  
    append(X, [H], Y).
```

This recursive case says that the reverse of a non-empty list can be found by reversing the tail of the list, and then appending a list consisting only of the first element of the original to the result of reversing the tail.

This predicate definition seems to be missing :

```
append([H|X], Y, [H|Z]) :-  
    append(X, Y, Z).  
append([], L, L).
```

The first clause is the recursive clause.

The second clause says that the result of appending an empty list with the list L is the list L itself.

Figure 7-6 : An incorrect algorithm match of a reverse/2 program by APROPOS2

Figure 7-7 shows another example of a reverse/2 program which has an additional clause of reverse([], Y) which improves the score of the match to the railway-shunt algorithm, and for which APROPOS2 can suggest the correct algorithm fit. This clause seems to improve on the fit to the railway-shunt reverse. The program analyses shows that in this instance, APROPOS2 was able to recognise the algorithm even though the student had missed out one argument.

Your program clauses are :

```
reverse([], []).
reverse([], Y).
reverse([H|T], Y) :-
    reverse(T, [H|Y]).
```

Your program seems to be incorrect.
Now, this is APROPOS2's critique on your program :

Clause 1 of reverse/2 seems to be redundant.

Clause 2 of reverse/2 does not seem quite right.
Replace reverse([], Y) with reverse([], Y, Y).
This clause instantiates the accumulated result to the result variable.

Clause 3 of reverse/2 does not seem quite right.
Replace reverse([H|T], Y) with reverse([H|T], Y, X)
and reverse(T, [H|Y]) with reverse(T, [H|Y], X).
This is what your clause should look like :

```
reverse([H|T], Y, X) :-
    reverse(T, [H|Y], X).
```

This clause uses an argument to accumulate the result as recursion proceeds, and another argument to be instantiated to the accumulated result when recursion stops.

This predicate definition seems to be missing :

```
reverse(X, Y) :-
    reverse(X, [], Y).
```

reverse/2 calls another auxiliary predicate reverse/3 to do the job.

Figure 7-7 : A correct algorithm match of a reverse/2 program by APROPOS2

In retrospect, since the frequency of programs having this clause :

```
reverse([H|T], Y) :-
    reverse(T, [H|Y]).
```

is high in our sample of student programs, it may be worthwhile to add a representation of this inadequate "algorithm" as another algorithm in the library of reference P-frames for reverse/2. This will then minimise the potential of matching the naive/inverse-naive/railway-shunt algorithms to these programs and will be able to provide more pertinent advice concerning the misconceptions involved.

Figure 7-8 shows some of the programs for which it is unclear what algorithms, if any, the student is intending when the program was written. For these programs, APROPOS2 can suggest an algorithm fit and the necessary changes to make the program work. For these cases, disambiguating the actual algorithm (or none) the student intended would be better handled by a student modelling component.

Program 1 :
reverse([], []).
reverse([HX|X], Y) :-
 append(Y, HX).

Program 2:
reverse([], []).
reverse([H1|T1], [H2|T2]) :-
 reverse(T1, T2),
 H2 is H1.

Program 3 :
reverse([], Y).
reverse([XH|XT], Y) :-
 reverse(XT, Y),
 append([Y|XH]).
append(X).

Figure 7-8 : Some student reverse/2 programs whose algorithms were unclear

It is possible that the student who wrote Program 3 has intended the naive algorithm. Program 3 indicates that he has a misconception that his append/1 clause allows a list and an element to be appended.

7.3 The replace_elements/4 task

The first of 3 programming tasks given to the MSc Prolog class of 1986/87 is replace_elements/4. The task statement is :

Task : replace_elements/4

Write a Prolog program replace_elements/4 which replaces all occurrences of a given element in a list by another and instantiates a given variable to the answer. The arguments should be, in order :

- 1) the element to be replaced
 - 2) what to replace it with
 - 3) the given list
 - 4) a variable to be instantiated to the answer.
- Your predicate should not bother to delve inside lists within lists.
-

There were 3 algorithms for the task represented as P-frames used for the analysis. They are described in Appendix I.2. The results of the analyses on all 22 submitted replace_elements/4 student programs are :

Total number of students' programs for replace_elements/4 :	22
No. of correct programs	19
No. of correct programs judged as correct by APROPOS2	19
No. of correct programs judged as incorrect by APROPOS2	0
No. of incomplete analyses by APROPOS2	0
No. of incorrect programs	3
No. of incorrect programs judged as correct by APROPOS2	0
No. of incorrect programs judged as incorrect by APROPOS2	3
No. of incomplete analyses by APROPOS2	0
Algorithm Recognition :	
No. of programs whose algorithms are correctly identified by APROPOS2	21
No. of programs whose algorithms are incorrectly identified by APROPOS2	1
No. of programs whose algorithms are unclear	0
Bug Detection and Correction :	
Total no. of bugs	5
Total no. of bugs detected and corrected	5
Total no. of bugs detected and corrected wrongly	0
Total no. of bugs not detected	0
Total no. of false alarms	0

Table 7-4 : Results of APROPOS2's analyses of students' replace_elements/4 programs

Of the 3 incorrect student programs, APROPOS2 recognises the algorithm for two of them. Here is one of them :

```

replace_elements(_, _, [], []).
replace_elements(EI, Rep, [F|R], [NF|NR]) :-
    replace_one(EI, Rep, F, NF),
    replace_elements(EI, Rep, R, NR).

replace_one(EI, Rep, EI, Rep) :-
    !.
replace_one(EI, Rep, [F|R], Result) :-
    !,
    replace_elements(EI, Rep, [F|R], Result).
replace_one(_, _, Atom, Atom).

```

APROPOS2's program critique is that the second clause of `replace_one/4` is extraneous and should be removed. This is the bug detected and corrected. The program delves into lists within lists which was not asked for. Although APROPOS2 finds the clause to be extraneous, it was not able to know that the purpose of the clause is to delve into lists within lists. This is because no knowledge of a clause to do this has been built into it.

Figure 7-9 shows the analysis of the other incorrect program whose algorithm is recognised by APROPOS2. In the analysis of this program, there is a potential for the code-matching process to be led astray because of the `write/1` subgoals. This did not turn out to be a problem as APROPOS2's code-matching process attributes a score of 0 when matching subgoals with side-effects such as `write/1` to `NOMATCH` for pure list and number manipulation tasks. However, for analysing other tasks, for example those that require failure-driven loops which use side-effecting subgoals, we cannot score these subgoals as 0.

Your program clauses are :

```

replace_elements(Old, New, [], []).
replace_elements(Old, New, [Old|T], Answr) :-
    write(Old),
    write(New),
    write(T),
    write([New|Answr]),
    nl,
    replace_elements(Old, New, T, [New|Answr]).
replace_elements(Old, New, [H|T], Answr) :-
    write(Old),
    write(New),
    write(T),
    write([H|Answr]),
    nl,
    replace_elements(Old, New, T, [H|Answr]).

```

Your program seems to be incorrect.
Now, this is APROPOS2's critique on your program :

Dataflow error in argument 4 of replace_elements/4

There will give you an incorrect computation as this argument will build up into a larger structure during recursive calls. The recursive subgoal call will never match the base case.

Clause 1 seems fine.

This base case says that the result of replacing elements in the empty list is the empty list.

Clause 2 does not seem quite right.

Replace `replace_elements(Old, New, [Old|T], Answr)` with
`replace_elements(Old, New, [Old|T], [New|X])` and
replace `replace_elements(Old, New, T, [New|Answr])` with
`replace_elements(Old, New, T, X)`.

`write(Old)` seems to be redundant.

`write(New)` seems to be redundant.

`write(T)` seems to be redundant.

`write([New|Answr])` seems to be redundant.

This is what your clause should look like :

```
replace_elements(Old, New, [Old|T], [New|X]) :-  
    replace_elements(Old, New, T, X).
```

This recursive clause handles the case where the head of the list in the third argument is identical to the element to be replaced.

Clause 3 does not seem quite right.

Replace `replace_elements(Old, New, [H|T], Answr)` with
`replace_elements(Old, New, [H|T], [H|X])` and
replace `replace_elements(Old, New, T, [H|Answr])` with
`replace_elements(Old, New, T, X)`.

`write(Old)` seems to be redundant.

`write(New)` seems to be redundant.

`write(T)` seems to be redundant.

`write([H|Answr])` seems to be redundant.

This is what your clause should look like :

```
replace_elements(Old, New, [H|T], [H|X]) :-  
    replace_elements(Old, New, T, X).
```

This recursive clause handles the case where the head of the list in the third argument is NOT identical to the element to be replaced.

Figure 7-9: APROPOS2's correct algorithm identification of a `replace_elements/4` program

Figure 8-10 shows the analysis of the incorrect program for which APROPOS2 did not find the best algorithm fit for the program. The better fit to the program would be this (Algorithm 1 of the algorithms listed in Appendix I.2) :

```
replace_elements(_, _, [], []).  
replace_elements(Element, New_element, [Element|Tl], [New_element|Tl]) :-  
    !,  
    replace_elements(Element, New_element, Tl, Tnl).  
replace_elements(Element, New_element, [H|Tl], [H|Tnl]) :-  
    replace_elements(Element, New_element, Tl, Tnl).
```

The second student clause has a missing recursive subgoal and as one matching heuristic used by APROPOS2 is to match student recursive clauses with reference recursive clauses, this seems to have misled the code-matching process.

Your program clauses are :

```
replace_elements(_, _, [], []).
replace_elements(Element, New_element, [Element|Tl], [New_element|Tl]) :-
    !.
replace_elements(Element, New_element, [H|Tl], [H|Tnl]) :-
    replace_elements(Element, New_element, Tl, Tnl).
```

Your program seems to be incorrect.

Now, this is APROPOS2's critique on your program :

Clause 1 seems fine.

This base case says that the result of replacing elements in the empty list is the empty list.

Clause 2 seems to be redundant.

Clause 3 does not seem quite right.

Replace `replace_elements(Element, New_element, [H|Tl], [H|Tnl])` with
`replace_elements(Element, New_element, [H|Tl], [X|Tnl])` and
a subgoal like `replace_one(Element, New_element, H, X)` seems to be missing.

This is what your clause should look like :

```
replace_elements(Element, New_element, [H|Tl], [X|Tnl]) :-
    replace_one(Element, New_element, H, X),
    replace_elements(Element, New_element, Tl, Tnl).
```

This recursive clause handles two cases : where the head of the list in the third argument is identical to the element to be replaced, and where the head of the list in the third argument is NOT identical to the element to be replaced.

This predicate definition seems to be missing :

```
replace_one(Old, New, Old, New) :-
    !.
replace_one(Old, New, Same, Same).
```

This first clause handles the case where the third element is identical to the element to be replaced. The second clause handles the case where the third element is NOT identical to the element to be replaced.

Figure 7-10 : APROPOS2's incorrect algorithm identification of a `replace_elements/4` program

7.4 The atom_count/2 task

The second of the 3 tasks is atom_count/2. The task statement is :

Task : atom_count/2

Write a Prolog program atom_count/2 which accepts a list as first argument and instantiates the variable given as second argument to the number of atoms in the list. The built-in predicate atom/1 succeeds if and only if its argument is an atom. Example :
:- atom_count([a, a, [b,a], []], Res).
Res= 3
% ie. the first two "a"s and the [] (which is an atom, as well as a list).

There are 4 algorithms represented in APROPOS2 for atom_count/2. The algorithms are described in Appendix I.3. The results of the analyses on all 22 submitted atom_count/2 student programs are :

Total number of students' programs for atom_count/2 :	22
No. of correct programs	19
No. of correct programs judged as correct by APROPOS2.....	18
No. of correct programs judged as incorrect by APROPOS2	1
No. of incomplete analyses by APROPOS2.....	0
No. of incorrect programs.....	3
No. of incorrect programs judged as correct by APROPOS2	0
No. of incorrect programs judged as incorrect by APROPOS2.....	3
No. of incomplete analyses by APROPOS2.....	0
Algorithm Recognition :	
No. of programs whose algorithms are correctly identified by APROPOS2	21
No. of programs whose algorithms are incorrectly identified by APROPOS2.....	1
No. of programs whose algorithms are unclear	0
Bug Detection and Correction :	
Total no. of bugs	6
Total no. of bugs detected and corrected	6
Total no. of bugs detected and corrected wrongly.....	0
Total no. of bugs not detected.....	0
Total no. of false alarms	0

Table 7-5 : Results of APROPOS2's analyses of students' atom_count/2 programs

Figure 7-11 shows APROPOS2's analysis of an incorrect program.

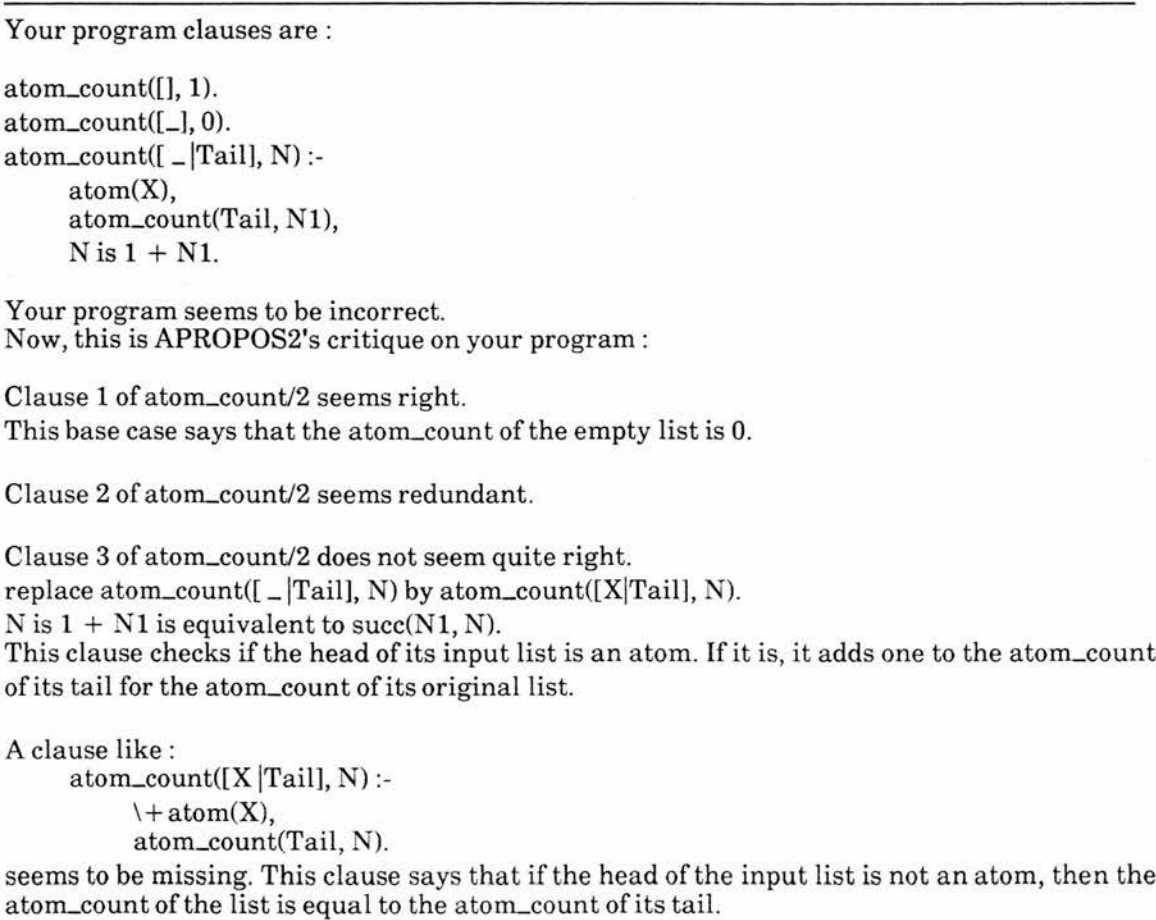


Figure 7-11 : APROPOS2's analysis of a student's atom_count/2 program

The critique on the second clause of atom_count/2 does not address the incorrectness of the clause. A more apt critique for the second clause of atom_count/2 would be :

Clause 2 of atom_count/2 seems incorrect.
It should be removed.

For APROPOS2 to provide this critique for base clauses, we can represent several bases cases in the task knowledge, some of which will be redundant. If the student base case is not one of these, then APROPOS2 can check its declarative reading by running it on some test cases and comparing the results obtained by running the same test cases on the reference program.

Figure 7-12 shows the analysis of the program whose algorithm is incorrectly identified by APROPOS2. APROPOS2 cannot match the student's atom_count/2 to any of the algorithms for the main task predicate, so it matches the next student predicate definition, namely

atomcount/2 to the algorithms for the main task predicate instead and finds a reasonable match with one of its reference algorithms. APROPOS2's analysis fails here as it cannot interpret what atomincrement/3 does. This student's program does give the correct answer. The second argument of atomincrement/2 is used to accumulate an unevaluated expression of the form $0 + 1 + \dots + 1$ where a 1 is added whenever the first argument of atomincrement/2 is an atom. When the top-level call of atomcount/2 succeeds, its second argument Numberlist will be instantiated to this unevaluated expression. Subsequently, the next subgoal

N is Numberlist

will evaluate the expression in Numberlist to get the desired result. APROPOS2 did not recognise the algorithm used as its program transformation is still primitive and it was unable to fold atomincrement/2 into atomcount/2. If it could do this, it would suggest a fix like replacing

```
atomincrement(Data, N + 1, N) :-  
    atom(Data).
```

by

```
atomincrement(Data, M, N) :-  
    atom(Data),  
    succ(N, M).
```

An in-depth dataflow analysis would be required for an automated debugger to recognise that the student's program does produce the right answer even though on subsequent forced backtracking, it incorrectly produces more incorrect answers.

Your program clauses are :

```
atom_count(List, N) :-  
    atomcount(List, Numberlist),  
    N is Numberlist.  
atomcount([], 0).  
atomcount([Head|Tail], Number) :-  
    atomcount(Tail, Number2),  
    atomincrement(Head, Number, Number2).  
atomincrement(Data, N + 1, N) :-  
    atom(Data).  
atomincrement(Data, N, N).
```

Your program seems to be incorrect.

Now, this is APROPOS2's critique on your program :

Your clause

```
atom_count(List, N) :-  
    atomcount(List, Numberlist),  
    N is Numberlist.  
can be rewritten as :  
atom_count(List, Numberlist) :-  
    atomcount(List, Numberlist).
```

Does not know what clause 1 of atomincrement/3 does.

Does not know what clause 2 of atomincrement/3 does.

Clause 1 of atomcount/2 seems fine.

This clause says that the atom_count of the empty list is 0.

Well, clause 2 of atomcount/2 does not seem quite right.

In clause 2 of atomcount/2 replace atomcount(Tail, Number2) with atomcount(Tail, Number).

In clause 2 of atomcount/2,

atomincrement(Head, Number, Number2) seems to be redundant.

This is what your clause should look like :

```
atomcount([_ |Tail], Number) :-  
    atomcount(Tail, Number).
```

This clause says that the atom_count of its input list is the atom_count of its tail.

In your predicate definition of atomcount/2, a clause like :

```
atomcount([X|Y], Z) :-  
    atom(X),  
    atomcount(Y, Z),  
    succ(V, Z).
```

seems to be missing. It should come before clause 2 of atomcount/2.

This clause checks if the head of its input list is an atom. If it is, add 1 to the atom_count of its tail.

Clause 1 of atom_count/2 seems redundant.

Figure 7-12 : APROPOS2's incorrect algorithm identification of an atom_count/2 program

7.5 The fringe_length/2 Task

The third of the 3 tasks is fringe_length/2. The task statement is :

```
Task : fringe_length/2
Write a Prolog program fringe_length/2 which expects a list as its first argument, and which
instantiates the variable given as its second argument to the number of elements in the list,
or in any lists in the lists, or in any lists in ... in the lists. If an element appears more than
once, it counts more than once. A list does not count as an element : instead it contributes its
fringe_length (which is 0 for an empty list) to the total. Examples :
:- fringe_length([a, a, [b, a]], Res).
Res= 4
% ie. a, a, b and a
:- fringe_length([a, [[a], []], [b,c]], Res).
Res= 4
% ie. a, a, b and c
:- fringe_length([[[[]], []], []], Res).
Res = 0
```

There are 5 algorithms represented in APROPOS2 for the task. The algorithms are described in Appendix I.4. The results of the analysis on all 22 submitted fringe_length/2 students' programs are :

Total number of students' programs for fringe_length/2 :.....	22
No. of correct programs	14
No. of correct programs judged as correct by APROPOS2.....	14
No. of correct programs judged as incorrect by APROPOS2	0
No. of incomplete analyses by APROPOS2.....	0
No. of incorrect programs.....	8
No. of incorrect programs judged as correct by APROPOS2	0
No. of incorrect programs judged as incorrect by APROPOS2	7
No. of incomplete analyses by APROPOS2.....	1
Algorithm Recognition :	
No. of programs whose algorithms are correctly identified by APROPOS2	20
No. of programs whose algorithms are incorrectly identified by APROPOS2.....	1
No. of programs whose algorithms are unclear	0
Bug Detection and Correction :	
Total no. of bugs	14
Total no. of bugs detected and corrected	14
Total no. of bugs detected and corrected wrongly.....	0
Total no. of bugs not detected.....	0
Total no. of false alarms	0

Table 7-5 : Results of APROPOS2's analyses of students' fringe_length/2 programs

Figure 7-13 shows APROPOS2's analysis of an incorrect program which is almost correct except for a missing empty list test in the second clause.

Your program clauses are :

```
fringe_length([], 0).
fringe_length([X|L], N) :-
    atomic(X),
    fringe_length(L, N1),
    N is N1 + 1.
fringe_length([X|L], N) :-
    fringe_length(X, NX),
    fringe_length(L, NL),
    N is NX + NL.
```

Your program seems to be incorrect.

Now, this is APROPOS2's critique on your program :

Clause 1 of fringe_length/2 seems fine.

This base case says that the fringe_length of the empty list is 0.

Clause 2 of fringe_length/2 does not seem quite right.

A subgoal like $X \neq []$ seems to be missing.

This is what your clause should look like :

```
fringe_length([X|L], N) :-
    atom(X),
    X \= [],
    fringe_length(L, N1),
    N is N1 + 1.
```

This recursive clause tests if the first element of the list is an atom and not the empty list. If so, find the fringe_length of its tail and add 1 to it.

Clause 3 of fringe_length/2 seems fine.

This recursive clause finds the fringe_lengths of the head and the tail of the list, and adds them up.

Figure 7-13 : APROPOS2's analysis of a student's fringe_length/2 program

Figure 7-14 shows the analysis of a program which uses an algorithm which we have missed out when we built the reference P-frames in the library for fringe_length/2. The algorithm first flattens the list and then counts the atomic elements. For completeness, we have included this algorithm the algorithms listed in Appendix I.4 even though we have not yet incorporated it into APROPOS2.. This is also the one incomplete analysis by APROPOS2. APROPOS2 fails in its analysis as it has no representation of the algorithm used. The student program is incorrect as it will count the empty list in the list as one in the second clause of fringe_length/2.

Your program clauses are :

```
fringe_length([], 0).
fringe_length([X|L], Answr) :-
    fringe_length(L, Answr1),
    atom(X),
    Answr is Answr1 + 1.
fringe_length([X|L], Answr) :-
    \+ atom(X),
    conc(X, L, Conc),
    fringe_length(Conc, Answr).
conc([], L, L).
conc([X|L1], L2, [X|L3]) :-
    conc(L1, L2, L3).
```

Your program seems to be incorrect.
Now, this is APROPOS2's critique on your program :

- Do not know what clause 1 of conc/3 does.
- Do not know what clause 2 of conc/3 does.
- Do not know what clause 1 of fringe_length/3 does.
- Do not know what clause 2 of fringe_length/3 does.
- Do not know what clause 3 of fringe_length/3 does.

Figure 7-14 : APROPOS2's incomplete analysis of a student's fringe_length/2 program

The program for which APROPOS2 incorrectly identifies the intended algorithm has a similar structure. The program uses the same algorithm as that in Figure 7-14. Figure 7-15 shows the analysis of the program.

Your program clauses are :

```
append([], L, L).
append([X|L1], L2, [X|L3]) :-
    append(L1, L2, L3).

fringe_length([], 0).
fringe_length([Head|Tail], N) :-
    \+ atom(X),
    append(Head, Tail, Ntail),
    fringe_length(Ntail, N1).
fringe_length([Head|Tail], N) :-
    atom(Head),
    fringe_length(Tail, N1),
    N is N1 + 1.
```

Your program seems to be incorrect.
Now, this is APROPOS2's critique on your program :

Clause 1 of fringe_length seems fine.
This base case says that the fringe_length of the empty list is 0.

Clause 2 of fringe_length/2 seems to be redundant.

Clause 3 of fringe_length/3 does not seem quite right.
A subgoal like Head\= [] seems to be missing.
This is what your clause should look like :

```
fringe_length([Head|Tail], N) :-
    atom(Head),
    Head\= [],
    fringe_length(Tail, N1),
    N is N1 + 1.
```

This recursive clause tests if the first element of the list is an atom and not the empty list. If so, find the fringe_length of its tail and add 1 to it.

A clause like :
fringe_length([X|Y], Z) :-
 \+ atom(X),
 fringe_length(X, W),
 fringe_length(Y, Y1),
 Z is W + Y1.

seems to be missing. This recursive clause finds the fringe_lengths of the head and the tail of the list, and adds them up.

Do not know what clause 1 of append/3 does.
Do not know what clause 2 of append/3 does.

Figure 7-15 : APROPOS2's failure to recognise the algorithm in a student's fringe_length/2 program

7.6 The sorting/2 task

Two students from the AI-2 class of 1987/88 tried the sorting/2 task interactively. We have described the analysis of the first student's program in Section 5.7. In this section, we describe the second student's session with APROPOS2. Figure 7-16 shows the task and student's first attempt at solution :

Task : sorting/2

Write a Prolog program sorting/2 which takes a list of numbers as input and sorts the numbers in the list in ascending order. A goal call "sorting(X, Y)" where X is the input list should succeed with Y instantiated to the sorted list. Use the predicate name sorting instead of sort which is an Edinburgh Prolog built-in predicate.

```
sorting([X|T], L) :-
    insert(X, T1),
    sorting(T, T1).
sorting([], []).

insert(X, [H|T], L) :-
    X < H,
    append([X], [H|T], L).
insert(X, [H|T], L) :-
    X > H,
    append([H], L1, L),
    insert(X, T, L1).
insert(X, [], [X]).
```

Figure 7-16 : A student's first attempt at sorting/2

Figure 7-17 shows APROPOS2's analysis of this student program.

Well, in this clause :
sorting([X|T], L) :-
 insert(X, T1),
 sorting(T, T1).
sorting/2 calls insert/2 which is not defined.

In this clause :
insert(X, [H|T], L) :-
 X < H,
 append([X], [H|T], L).
insert/3 calls append/3 which is not defined.

Do you want to edit your file now (If no, analysis continues) ?**no**

In the clause :
sorting([X|T], L) :-
 insert(X, T1),
 sorting(T, T1).
L only appears once.

Your clause :
insert(X, [H|T], L) :-
 X < H,
 append([X], [H|T], L).

can be rewritten as :
insert(X, [H|T], [X, H|T]) :-
 X = < H.

Your clause :
insert(X, [H|T], L) :-
 X > H,
 append([H], L1, L),
 insert(X, T, L1).

can be rewritten as :
insert(X, [H|T], [H|L1]) :-
 X > H,
 insert(X, T, L1).

Do you want to the analysis to proceed ?yes

Your program seems to be incorrect.
Now, this is APROPOS2's critique on your program :

Clause 1 of insert/3 seems fine.
This clause handles the case when the element to be inserted is smaller than the first element of the list to be inserted into. It therefore puts the element to the front of the list.

Clause 2 of insert/3 seems fine.
This clause handles the case when the element to be inserted is larger than the first element of the list to be inserted into.

Clause 3 of insert/3 seems fine.
This base case says that the result of inserting an element X into the empty list is the list [X].

Clause 1 of sorting/2 does not seem quite right.
Replace insert(X, T1) with insert(X, T1, L).
This recursive case sorts the tail of the list and insert the first element back into the sorted tail. But you should place the recursive subgoal before the insert/3 subgoal.

Clause 2 of sorting/2 does not seem quite right.
Replace sorting([X], [X]) with sorting([], []).
This base case says that the result of sorting the empty list is the empty list.

Figure 7-17 : APROPOS2's analysis on a sorting/2 program

In his next attempt, the student made the following changes to his program :

- insert(X, T1) is replaced with insert(X, T1, L), and
- a predicate definition for append/3 was added.

One observation here is that the student did not follow the advice given by APROPOS2 in toto. This also indicates the difficulty of evaluating efficacy of systems like APROPOS2. Such evaluation have to take into account factors such as the number of students who follow the advice given, those who ignore the advice and those who heed only part of it.

The relevant part of APROPOS2's critique of this modified program is :

Clause 1 of `sorting/2` does not seem quite right.

This recursive case sorts the tail of the list and insert the first element back into the sorted tail. But you should place the recursive subgoal before the `insert/3` subgoal.

APROPOS2 ran the student program on an example `sorting([2,3,1,4],X)`, and from the execution, found some execution errors of "uninstantiated variables in arithmetic expression" and that the student program produced no solutions. In his third attempt, the student made the following changes to his program :

- the two insert clauses were rewritten by folding the append subgoal away,
- swapped the order of the insert and recursive subgoals in the `sorting/2` recursive clause.

APROPOS2 then accepted this program as correct.

7.7 An Independent Judgement of APROPOS2's Analyses

In the above discussion, the decision of whether APROPOS2 has identified an algorithm correctly, detected the bugs and suggested the bug corrections is based on the author's own judgement. To seek an independent evaluation, we asked a Prolog lecturer in the Department of Artificial Intelligence to go through APROPOS2's analyses of the 95 `reverse/2` programs *and* our judgement of these analyses, and to comment upon them. As the Prolog lecturer has been teaching Prolog to students for 6 years, we believe he would be a fair judge of APROPOS2's performance. Appendix V shows the form which was given to the Prolog lecturer. It is intended to give the reader a clearer idea about APROPOS2's analyses and the kind of test we use to evaluate the analyses.

We concur in our judgement of 85 programs. We agree on the intended algorithm used in these programs, and what are the bugs in the incorrect programs. We interpret the other 10 programs somewhat differently. We have considered these programs as intending to use the naive algorithm.

Program 1 (4 programs in all) :

```
reverse([], []).
reverse([H|T1], [T2|H]) :-
    reverse(T1, T2).
```

Program 2:

```
reverse([], Y).
reverse([H|T1], [T2|H]) :-
    reverse(T1, T2).
```

```

Program 3 :
reverse([], []).
reverse([N], [N]).
reverse([A|B], [C|A]) :-
    reverse(B, C).

```

```

Program 4:
reverse([], []).
reverse([X], [X]).
reverse([Head|Tail], [Contents_of_reversed_tail, Head]) :-
    reverse(Tail, [Contents_of_reversed_tail]).

```

```

Program 5 :
reverse([], L).
reverse([H], Ans) :-
    Ans = [Ans|H].
reverse([H|T], Ans) :-
    Ans = [Ans|H],
    reverse(T, Ans).

```

```

Program 6 :
reverse([], []).
reverse([Head|Tail], [X|Head]) :-
    reverse(X, Tail).

```

Figure 7-18 : reverse/2 programs written by students with buggy models of Prolog

The Prolog lecturer saw these programs as written with buggy models of Prolog, especially pertaining to Prolog syntax, and therefore can be addressed without reference to what algorithm is intended. We agreed that these programs exhibit buggy models of Prolog. However, we also interpret these programs as attempting to reverse the tail of the list and splice the head element of the list into the reversed tail to get the desired result. Therefore treating this as the intention of the programmer, we consider the programs as intending to implement the naive method. APROPOS2 detects algorithms in all these programs as its code analysis is algorithm-based although it may sometimes fail to match any of the algorithms it knows of to programs which differ too much from the possible implementations of the algorithms. For Programs 1 to 5 in Figure 7-18, APROPOS2 detected the naive method, while for Program 6, it chose the best fit from the railway-shunt method.

The other program for which we differ in judgement is :

```

Program 7 :
reverse([], []).
reverse([], Y).
reverse([H|T], [X]) :-
    reverse(T, [X|H]).

```

Figure 7-19 : reverse/2 program written by student whose algorithm is unclear

We consider this as intending the railway-shunt algorithm. The Prolog lecturer's observations are that the algorithm is unclear and the student has a buggy model of dataflow. He commented : *"The reason is that the only basis for guessing the student intends the railway-shunt algorithm is that the second argument seems to be more complex in the recursive subgoal. But the second argument has been assembled from peculiar components."* We concurred that the student who wrote this program had a buggy model of dataflow in Prolog, as well as problems with Prolog syntax.

In this independent evaluation, we provided the Prolog lecturer with a summarised account of APROPOS2's analyses. The reason for including APROPOS2's analyses was to facilitate the task of the independent reviewer who had to pore over 55 different programs. This is not as ideal as we have wished. We would have preferred to carry out a more objective evaluation in which the independent reviewer is given the set of student Prolog programs and the list of algorithms used in APROPOS2 but not APROPOS2's program analyses, and asked to judge the algorithm intended in each program, detect the bugs and suggest corrections. Since we have included a summary of every reverse/2 program analysis in Appendix V, the reader may like to judge for himself in his opinion the validity of APROPOS2's analysis and our judgement of it.

We did not seek an independent judgement of APROPOS2's analyses of the other tasks, but the high percentage of correct program recognition of the mostly correct programs (whose algorithms seem lucid enough) written for these tasks give us reason to believe that our evaluation of APROPOS2's analyses is objective enough in the main.

7.8 Discussion of Observations Drawn

In this section, we discuss what we have learned about the efficacy of APROPOS2's program analysis approach from our initial tests of its performance on the analysis of real student programs. Our main findings are :

- APROPOS2 can recognise the algorithms used in a high proportion of students' programs (85% for the reverse/2 task and 95% for the other tasks). We think that the main reason that account for APROPOS2's performance is the nature of the programming tasks. Tasks like reverse/2 and sorting/2 have standard algorithms for their computer implementations. Using representations of common implementations of these algorithms, we have been able to cover a large proportion of programs students actually wrote. We list the cases where APROPOS2's analysis fails :
 - (1) when an algorithm has not been represented in APROPOS2's task knowledge, APROPOS2 will either fail to do a complete analysis or will identify another

algorithm (as in the analyses shown in Figures 7-14 and 7-15). This illustrates the limitation of APROPOS2's approach. However, it must be said also if a human teacher cannot understand the algorithm used, he will not be able to criticise the correctness of the program.

- (2) when a frequently used but misconceived implementation of an algorithm has not been represented in the P-frame for the algorithm (as in reverse/2 programs that have a clause of :

```
reverse([H|T], Y) :-
    reverse(T, [H|Y]).
```

), APROPOS2's analysis may turn out to be inapt.

- (3) when a novel implementation of an algorithm is not present in its P-frame, APROPOS2's analysis may fail to recognise correct programs (as in the analysis in Figure 7-2).

While the dependency of APROPOS2's performance on its knowledge of algorithms and implementations of algorithms is a real limitation, it is not clear from the research literature what solutions have been proposed to address this limitation for other tutoring programs.

We see APROPOS2 as an evolving program. As it is put to more tests, post analysis of its performance will reveal the deficiencies of APROPOS2's task knowledge. Doing more of these tests should suggest what missing algorithms and what misconceived implementations of an algorithm, which must be observable in a small sample of the population of programs tested, to add to APROPOS2's task knowledge base. The issue of whether we should add novel implementations of an existing algorithm to the P-frame for the algorithm is less straightforward. It is not certain we should do this if the toms in all levels of a list

We evaluated APROPOS2 on the corpus of students' reverse/2 programs collected in the empirical trials of APROPOS1. APROPOS2 has also been put to the test on three of many programming tasks given to the

APROPOS2's performance on the tasks of replace_elements/4, atom_count/2, fringe_length/2 and to a certain extent for reverse/2, suggests that we have accumulated a critical mass of knowledge in the P-frames for APROPOS2 to be able to handle real programs students wrote for these tasks.

- If APROPOS2 can recognise the algorithm used in a student's program, it can detect a high proportion of the bugs and correct them. For the reverse/2 task, APROPOS2 gives false alarms in only two programs, while for the other tasks, APROPOS2 achieves full

accuracy in bug detection and correction. The key factor to APROPOS2's performance is thus its ability to judge an algorithm correctly.

- While APROPOS2's analyses may not be the most apt under the circumstances, it does suggest fixes that always result in correct solutions that evolve out of the incorrect student's program. This is at least useful for students who do not have a fair or any idea about writing a program for a task or who can only write a partial program. APROPOS2 will then be able to complete the program for them and provide associated commentary on the function of each clause.

We have seen how APROPOS2 handles the four tasks of `reverse/2`, `replace_elements/4`, `atom_count/2`, and `fringe_length/2` for a small number of actual student programs. This gives us reason to expect that APROPOS2 can also handle list-manipulation tasks like `member/2`, `last_element/2`, `sublist/2`, `delete/2`, `flatten/2`, `intersect/2`, `union/2`, etc. These are the kind of tasks novices typically spend some time during their initial learning writing programs for. Indeed, this further confirms the usefulness for a practical tool like APROPOS2.

One general observation about the programs students wrote is that they make all kinds of errors – slips, bad planning, nonconceptual errors, and errors involving deeper misconceptions. They fall into no fixed patterns and no consistent model of programming. This provides a good argument for using a code-matching approach. Programs vary in their implementations and this include some maverick ones, indeed illustrating the expressiveness of Prolog as a programming language. It will then be difficult to diagnose errors using a buggy model of problem-solving similar to that of [Burton 81] and [Coombs et al 86] which we have discussed in Section 2.5.1. A mistyping error can easily put their approach of bug diagnosis off.

In summary, our empirical tests with APROPOS2 shows that our debugging approach gives a reasonable standard of performance. In most cases, it identifies algorithms accurately in our judgement. When it does so, it identifies and corrects bugs accurately. Further enhancements of the task-knowledge base are expected to improve its performance.

7.9 Intent of Evaluation Study

In our evaluation of APROPOS2, we are not studying the educational effectiveness of using APROPOS2. Such an evaluation is another research project in itself. Sack describes some of the problems involved in testing the educational effectiveness of using the PROUST system : students only use PROUST on one programming problem, some might have got no advice from PROUST, some got bad advice, some got good advice but they didn't bother to look at the advice [Sack 87]. These suggest the kind of factors which need to be controlled in doing a

study which must involve upwards of 300 students in order that the sample population get broken down into significant subgroups.

APROPOS2 is developed in the process of a research project. It is never intended to be a finished product. Our evaluation of APROPOS2 is exploratory in as much as most of AI research is exploratory. We evaluated it as a prototype to test the feasibility of its program analysis approach. We carried out a *formative* evaluation, not a final, definitive, *summative* evaluation [Wagner & Seidel 78]. The objective of a formative evaluation is not to determine yes/no with respect to a given product, but rather to gather data on how/why the product is being used, which in turn, can be used to further develop the product [Sack et al 86]. In the next chapter, we bring together ideas on how we can further develop APROPOS2.

Chapter 8. Future Directions for APROPOS2

We have elsewhere in the thesis alluded to immediate extensions to APROPOS2. In this chapter, we consider broader directions for extending this work on APROPOS2. We will propose several issues for further research.

8.1 Analysing APROPOS2's Heuristic Code-Matching Process

APROPOS2 matches a student Prolog program to different implementations of an algorithm using a measure of closeness of match. This measure comprises scores from comparing features of the student program with features of the reference program. The scores for matching features and components of predicate definitions are obtained by considering how much of a misfit each match can contribute to the overall match. The scores have been tested and tuned on an initial set of reverse/2 collected programs. Our empirical tests on four programming tasks indicate a good level of performance of APROPOS2's algorithm recognition and bug detection. As part of future work on APROPOS2, we propose to study how sensitive is APROPOS2's performance to the choices of the scores. This work may involve perturbing students' programs to see how much of APROPOS2's critique is affected for each incremental perturbation.

8.2 Exploring a Generative Theory of Programming

APROPOS2's code-matching approach parses the student solution into a decomposition of predicate definitions, clauses, subgoals, arguments and terms. An interesting issue is whether we can use this as the basis for formulating a kind of "*generative theory*" of correct and buggy programs [van Someren 87]. We first find an algorithm, decompose the algorithm into the main and auxiliary predicate definitions, decompose the predicate definition into its component clauses, and then onto subgoals, arguments and finally terms. In all these processes, Prolog programming techniques are used. For example, techniques may be selected to decompose a predicate definition into its component clauses. The incorrect use of a technique or the use of an inappropriate technique may then give rise to bugs. This may go some way to help postulate the misconceptions that underly the bugs in students' programs.

8.3 Devising More Abstract DataFlow Representations

A global analysis of dataflow in any program usually involves representing the dataflow of the program in some form of representation such as graphs. We have not looked in any great depth into what form of abstract dataflow representation may be suitable for Prolog programs. Such a dataflow analysis should be able to show that all these three versions of the railway-shunt reverse have the same dataflow :

PROGRAM 1 :

```
reverse(X, Y) :-  
    reverse(X, [], Y).  
reverse([], L, L).  
reverse([A|X], Y, Z) :-  
    reverse(X, [A|Y], Z).
```

PROGRAM 2 :

```
reverse(A, B) :-  
    rev(A, [B]).  
rev([], [B|B]).  
rev([Head|Tail], [B|Temp]) :-  
    rev(Tail, [B,Head|Temp]).
```

PROGRAM 3 :

```
reverse(X, Y) :-  
    dreverse(X, Y-[]).  
dreverse([X|Xs], Ys-Zs) :-  
    dreverse(Xs, Ys-[X|Zs]).  
dreverse([], Xs-Xs).
```

Program 2 can be transformed to Program 1 by using the following transformation rule :

rewrite `rev(Arg1, [Arg2Head|Arg2Tail])` as `rev(Arg1, [Arg2Head], Arg2Tail)`.

`rev/2` is then rewritten as :

```
rev([], [B], B).  
rev([Head|Tail], [B], Temp) :-  
    rev(Tail, [B], [Head|Temp]).
```

As the second argument of both clauses, `[B]`, is of type list and a '-' mode, we can simplify `rev/2` to :

```
rev([], B, B).  
rev([Head|Tail], B, Temp) :-  
    rev(Tail, B, [Head|Temp]).
```

In a similar way, Program 3 can be transformed to Program 1 by using the following transformation rule :

rewrite `dreverse(Arg1, Arg2Front-Arg2Tail)` as `dreverse(Arg1, Arg2Front, Arg2Tail)`.

Currently, we have not implemented these transformation rules in APROPOS2. Thus it is not able to recognise the algorithm in Program 2. Incorporating these transformation rules is a difficult task as the number of such rules is as large as the number of ways of disguising the accumulator, which are numerous. One solution to this is to attempt to represent the abstract dataflow of a program for every algorithm in some canonical form, so that every program implementing the algorithm has the same dataflow. The advantage with this approach is that different implementations incorporating the same dataflow can be recognised. A disadvantage to this approach is that it will then be difficult to acquire information about the implementation of the particular algorithm the student used for writing the program and provide specific advice about the nature of the bugs and its underlying misconceptions, if any, that are involved. Nevertheless, devising abstract representations of algorithms would be another research issue in itself. It will be interesting to look into explicit representations of how Prolog programs iterate, construct or explode recursively-defined data structures and find out if how such abstract computational features of Prolog programs can be used to describe why a program works, or does not work, or why it is inefficient.

8.4 Further Reasoning with Modes

In this section, we propose how we can further extend the use of modes in analysing Prolog programs for debugging and tutoring purposes. We propose an approach to explaining the run-time behaviour of programs primarily in terms of modes.

Consider the standard definition of `append/3` :

```
append([], L, L).
append([H|L1], L2, [H|L3]) :-
    append(L1, L2, L3).
```

Let us consider how Lever's method [Lever 85] might infer the output mode from the input mode of `append(-, +, -)`. We first construct a partial input mode `append(X, +, Y)` by substituting variables for the '-' mode, and match with the recursive clause to give a clause of the form :

```
append([H|L1], +, [H|L3]) :-
    append(L1, +, L3).
```

with bindings $(X/[H|L1])$ and $(Y/[H|L3])$ created. Since the partial mode of the recursive call matches the partial mode of the initial call, Lever's method says that we can compute the output mode of the recursive call by matching its input mode with the base clause because of the ways the variables are used in this example. The idea is that when the modes of the initial and the first recursive call agree for all argument positions, all subsequent recursive calls

will have the same mode. Matching `append(L1, +, L3)` with `append([], L, L)` gives `append(+, +, +)` with no new bindings created. Finally, we derive the partial output mode for the initial call `append([H|+], +, [H|+])`, giving the output mode of `append([-|+], +, [-|+])`.

The output mode computed by Lever's method is conservative and is only true for the first solution returned. Clause ordering is important too, for example, if the base clause of `append/3` is placed after the recursive clause, then a mode of `append(-, +, -)` gives rise to nontermination. A call of mode `append(-, +, -)` is nondeterminate – it produces infinite solutions on backtracking.

The use of mode analysis can be further extended to help in explaining the running of a student's Prolog program. Consider this student's `reverse/2` program :

```
reverse(A, B) :-
    rev(A, [B]).
rev([], [B|B]).
rev([Head|Tail], [B|Temp]) :-
    rev(Tail, [B,Head|Temp]).
```

This student program is an implementation of the railway-shunting reverse using an accumulator. It disguises the result argument in the second argument of `rev/2`. How might we explain the running of this program? By dynamic analysis, we know that `reverse(+, -)` produces only one solution which is correct and `reverse(-, +)` produces one solution which is correct and then loops. To see this looping behaviour, a mode call of `reverse(-, +)` leads to a mode call `rev(-, +)` which leads successively to a mode call `rev(-, +)` which succeeds once only in the base case and on backtracking loops forever.

Consider this program `nrev/2` :

```
nrev([H|T], Res) :-
    append(Sofar, [H], Res),
    nrev(T, Sofar).
nrev([], []).
append([], L, L).
append([H|L1], L2, [H|L3]) :-
    append(L1, L2, L3).
```

Through some form of code-matching or theorem-proving, an automated debugger would be able to know that the clause is correct in its declarative reading. To see its run-time behaviour, we run the program on some test input on an interpreter with a goal stack limit to trap non-terminating computations and we force backtracking to see how many and what solutions it succeed with.

Suppose we attempt a goal call of mode `nrev(-, +)`. Then the instantiation states on invocation will be :

$\text{nrev}(-, +) :- \text{append}(-, +, +), \text{nrev}(-, -).$

A call of $\text{nrev}(-, +)$ leads to a call of $\text{append}(-, -, +)$ which succeeds to $\text{append}(+, +, +)$ as can be inferred from Lever's method. This leads to a recursive call of $\text{nrev}(-, +)$ which would match the base case at some point giving an output mode of $\text{nrev}(+, +)$. This will eventually produce an output mode of $\text{nrev}(+, +)$ for the initial call.

Strictly speaking, when a goal invocation of $\text{nrev}(-, +)$ gets matched to $\text{nrev}([H|T], \text{Res})$, it becomes $\text{nrev}([-], \text{Res})$ or $\text{nrev}(+, +)$. We interpret

$\text{nrev}(-, +) :- \text{append}(-, +, +), \text{nrev}(-, -).$

to mean a call of mode $\text{nrev}(-, +)$ leads to a call of mode $\text{append}(-, +, +)$ and later to a call of mode $\text{nrev}(-, -)$ whose instantiation state may depend partly on the state of instantiation of arguments of the append call on its success.

Suppose we attempt a goal call of mode $\text{nrev}(+, +)$. Then the instantiation states in the recursive clause on invocation will be :

$\text{nrev}(+, +) :- \text{append}(+, +, +), \text{nrev}(+, +).$

A goal call with mode $(+, +, \dots, +)$ will either succeed once only, fail or loop. Since the first argument of nrev is used as the iteration argument and gets smaller in list length in the recursive call, there is no potential for looping and $\text{nrev}(+, +)$ will therefore either succeed once only or fail.

Suppose we run a goal call of modes $\text{nrev}(+, -)$. It will produce one answer and then loop nonterminatingly. On invocation, the instantiation states of the arguments in the $\text{nrev}/2$ recursive clause will first be :

$\text{nrev}(+, -) :- \text{append}(-, +, -), \text{nrev}(+, -).$

On invocation of this clause, control goes to satisfying the append subgoal. A call of $\text{append}(-, +, -)$ will succeed with first and third arguments instantiated to a totally ground term $(+)$ or a term with uninstantiated components $([-|+])$ or simply $(+)$. Control flow then goes to the nrev recursive subgoal which will then be $\text{nrev}(+, +)$. Whether this succeeds or not, successive backtracking will attempt to resatisfy the append subgoal which always succeeds and instantiates components of its first argument. By dynamic analysis using a depth-bound interpreter, we know that $\text{reverse}(+, -)$ succeeds once and loops on backtracking. We say that the recursive subgoal $\text{nrev}(+T, -\text{Sofar})$ succeeds for only one value of Sofar because of its declarative semantics, and fails for all other values for Sofar ; on forced backtracking after producing an answer, this subgoal fails, control goes back to $\text{append}(-\text{Sofar}, +[H], -\text{Res})$

which generates infinite solutions for Sofar which the nrev recursive call always fails and rejects, thus resulting in a nonterminating computation.

Suppose we attempt a goal call of mode $nrev(-, -)$. Then the instantiation states on input will be :

$$nrev(-, -) :- \text{append}(-, +, -), nrev(-, -).$$

A call of $\text{append}(-, +, -)$ will succeed to $\text{append}(+, +, +)$ which then leads to a call of $nrev(-, +)$ which will succeed to $nrev(+, +)$. On successive backtracking, $\text{append}(-, -, -)$ produces more solutions and leads to more calls of the form $nrev(-, +)$ which will succeed to $nrev(+, +)$. So, $nrev(-, -)$ produces infinite solutions.

Consider this clause in which the order of the subgoals is reversed :

$$\begin{aligned} nrev([H|T], \text{Res}) :- \\ \quad nrev(T, \text{Sofar}), \\ \quad \text{append}(\text{Sofar}, [H], \text{Res}). \end{aligned}$$

Suppose we run a goal call of modes $nrev(-, +)$. It will produce one answer and then loop. The instantiation states of the arguments in the $nrev/2$ recursive clause on invocation will be :

$$nrev(-, +) :- nrev(-, -), \text{append}(-, +, +).$$

Control flow goes to satisfying the $nrev$ recursive call. If the $nrev/3$ recursive clause is placed after its base clause, then the $nrev$ recursive subgoal invocation will loop. If the base clause is placed first, then $nrev(-, -)$ will first succeed in matching the base case returning $nrev(+, +)$. Control flows to $\text{append}(+, +, +)$. Whether this succeeds or not, successive backtracking will attempt to resatisfy the $nrev(-, -)$ subgoal which will then match the recursive clause :

$$nrev(-, -) :- nrev(-, -), \text{append}(-, +, -).$$

This call produces infinite solutions for a call of $nrev(-, -)$. $\text{append}(-\text{Sofar}, +[H], +\text{Res})$ is determinate as can be inferred from a similar mode chain analysis using the $\text{append}/3$ predicate definition. So, $nrev(-T, -\text{Sofar})$ produces many solutions for Sofar but for which the append subgoal invocation only succeeds on one – another instance of *infinite generate and futile test*.

In the reference P-frame representations of APROPOS2, we use a slot to incorporate the constraint on clause ordering information. An example of a clause ordering constraint is that for the $\text{append}(-, +, -)$ P-frame, the recursive clause should come after the base clause otherwise a call of mode $\text{append}(-, +, -)$ will result in nontermination. Ideally, we want to alleviate the use of this slot and, in place of that, use mode analysis to reason about clause orderings, such as what we have discussed in this section.

In this section we have described how we might make use of mode reasoning to explain the run-time behaviour of Prolog programs. A question for further research is to what extent can we automate this reasoning process. Mode information may also be used to help in the analysis of clause ordering, subgoal ordering and determinacy of computations.

8.5 Utilising More Rigorous Sources of Program Analysis

We have utilised multiple sources of program analysis in APROPOS2. We have written a basic type inference program for handling some simple types. Our program transformation rules are very specific and cannot be guaranteed not to change the meaning of the program they are applied to. We check for dataflow anomalies which are indicative of program bugs. Some of these sources of program analysis can be further improved for more rigour. In this section, we briefly mention related work done in the logic programming community on analysing properties of logic programs.

Researchers in the logic programming community have recently studied the problem of presenting an unified framework for dataflow analysis of logic programs based on the techniques of *abstract interpretation* [Bruynooghe et al 86; Debray 86; Smolka 86]. They have adapted the seminal work of [Cousot & Cousot 77] on the abstract interpretation of imperative languages for logic programs. Abstract interpretation is a technique for the static analysis of programs based on the underlying program semantics. The notion is to finitely approximate the infinite computational domain of a program by defining equivalence classes over them, and simulating computations over these equivalence classes instead. By finding the fixpoint(s) over such approximate computations, it is possible to draw conclusions that describe approximately the runtime behaviour of programs. One venue of interesting research is to bring these abstract dataflow frameworks into practical use in analysing properties of Prolog programs.

In the logic programming community, work has been done on the derivation and checking of global properties of Prolog programs which may be brought into practical use for debugging programs for tutoring purposes. Bruynooghe [Bruynooghe 81] first broached the idea of adding type and mode information to any Prolog program so that the program can be checked to see if they are consistent with this information. Mellish's mode inference program for inferring invocation modes which we have used in APROPOS uses abstract interpretation. Mellish [Mellish 85] also uses an abstract interpretation approach to detect determinacy of predicates. Debray & Warren [Debray & Warren 86] describes a sufficiency test to determine which predicates are functional; their notion of functionality subsumes determinacy. The objective of this early work is to achieve code and compiler optimisation. Debray describes a unified framework for dataflow analysis of logic programs based on the techniques of abstract

interpretation [Debray 86], [Reddy 84], [Smolka 84], [Bruynooghe et al 86] and [Lever 87] studied aspects of modes and dataflow. [Mycroft & O'Keefe 84], [Mishra 84], [Kanamori & Horiuchi 85], [Bruynooghe et al 86], [Yardeni & Shapiro 87], [Kluzniak 87] and [Zobel 87] have looked into different aspects of type checking and type derivation. [Drabent & Maluszynski 86] proposes an approach to proving run-time properties of computations of logic programs, such as properties about modes, which is similar to the method of proving inductive assertions. One of the problems here is the difficulty in synthesizing inductive assertions. All these provide more formal analyses of modes, types and determinacy which might be incorporated into a later version of APROPOS for greater rigour.

Chapter 9. Future Directions for a PITS

In this chapter, we will discuss the integration of APROPOS2 in a complete PITS. We return to the overall context of building APROPOS2 – as part of the expertise module of a PITS. We describe one view of how PITS might interact with a student, and discuss the role we envisage APROPOS2 would play in our view of a PITS.

9.1 The Role of a PITS

Several researchers have alluded the problem of building a comprehensive ITS as equal to the task of the whole AI enterprise. The status of existing working systems indicates that a PITS can only do a small part of the job of a human teacher. Given the difficulties of the task, what part of the role of the Prolog teacher should we reasonably aim to automate? We concur with the view of Ross who believes that the single most troublesome area for Prolog novices is that of converting some problem statement into an algorithm after they have endured the first few weeks of cognitive disorientation when first learning about the things that are commonly done in that language [Ross 87]. We envisage a PITS which is practical and useful for the first few weeks in the learning phase of the novice, a PITS which can lead students through the cliché-practice stage, facilitating the learning process and allowing the Prolog teacher to spend more time on more open-ended problems. Being useful for a short period is also true for most other ITSs for programming with the exception of GREATERP [Reiser et al 85] which cover some 85% of the introductory LISP curriculum. It is possible for GREATERP to do this as it constrains the student to follow the method of solution it knows of and forbids otherwise.

We thus destine PITS for use in the early learning of Prolog novices, to relieve the Prolog teacher from spending too much time on criticising answers to simple exercises and let him concentrate more on the development of initiative at the problem analysis and program design stage. More precisely, PITS is intended for the student who has learnt the basics of Prolog syntax and semantics and who has some idea about the control flow.

We intend the mode of use of PITS to be interactive. Task statements are posed to the student and the student is asked to write predicate definitions for the task. An important part of PITS is a program debugger that diagnoses bugs in the student program and suggests bug fixes. The location, description and correction of all bugs will then be used by the student modelling component to infer possible misconceptions the student may have. All this information can

then be used by the tutorial expert to provide remedial instruction to resolve the misconceptions.

9.2 The Role of APROPOS2 in a PITS

A complete ITS for teaching programming includes a domain expert, a student modeller and a tutorial expert. The expertise module or domain expert consists of domain knowledge that the system intends to teach the student. It is used to generate instructional content and to evaluate the student's performance. The student model is used to assess the student's knowledge state and to make hypotheses about his or her conceptions and programming knowledge employed to achieve the current knowledge state. The tutorial module specifies what instructional material the system should present and how and when it should present it.

Students learn programming by having to do it themselves. A PITS needs to be able to provide programming problems to students, observe them as they write their programs as well as evaluate their programs. We suggest one way how a PITS might interact with a student. PITS provides a problem statement to the student. The student can ask for examples of test cases, write the program for the problem, or ask for help or hints. If help is asked for, PITS might provide a description of an algorithm for the task, for example, one whose implementation requires the use of some programming skills that the student is not believed to have acquired but would be able to learn from. PITS accepts a whole program in the form of the main predicate and all the auxiliary ones. The student can request PITS's critique on a partially finished program in which some auxiliary predicates are not yet written. PITS will generate analyses of the student's input to assess its correctness either in itself or as part of a program that solves some task. It selects the best analysis and comments on it. The session is intended to be interactive, the student can be asked questions to confirm PITS's hypotheses, or choose to run his program or parts of it himself.

We intend APROPOS2 to be an important part of the expertise module of a PITS. Figure 9-1 shows the role of APROPOS2 in a PITS. In our view of a PITS, the user interface presents instructional material to the student and gets some input from the student. The interface can incorporate a structure editor for constructing Prolog programs. The input may be a predicate definition or a complete program or something in between. APROPOS2 receives student code from the user interface as input and debugs this code. It produces a record of the best few analyses of the student's code with records of the detected bugs and the bug corrections. This information is used by the student modeller to select which analysis is the most likely and the possible misconceptions that the student may have. Selecting the best analysis may be assisted by asking the student via the user interface. The tutorial expert will use all these information to choose a teaching strategy to resolve the misconceptions. Remedial instruction

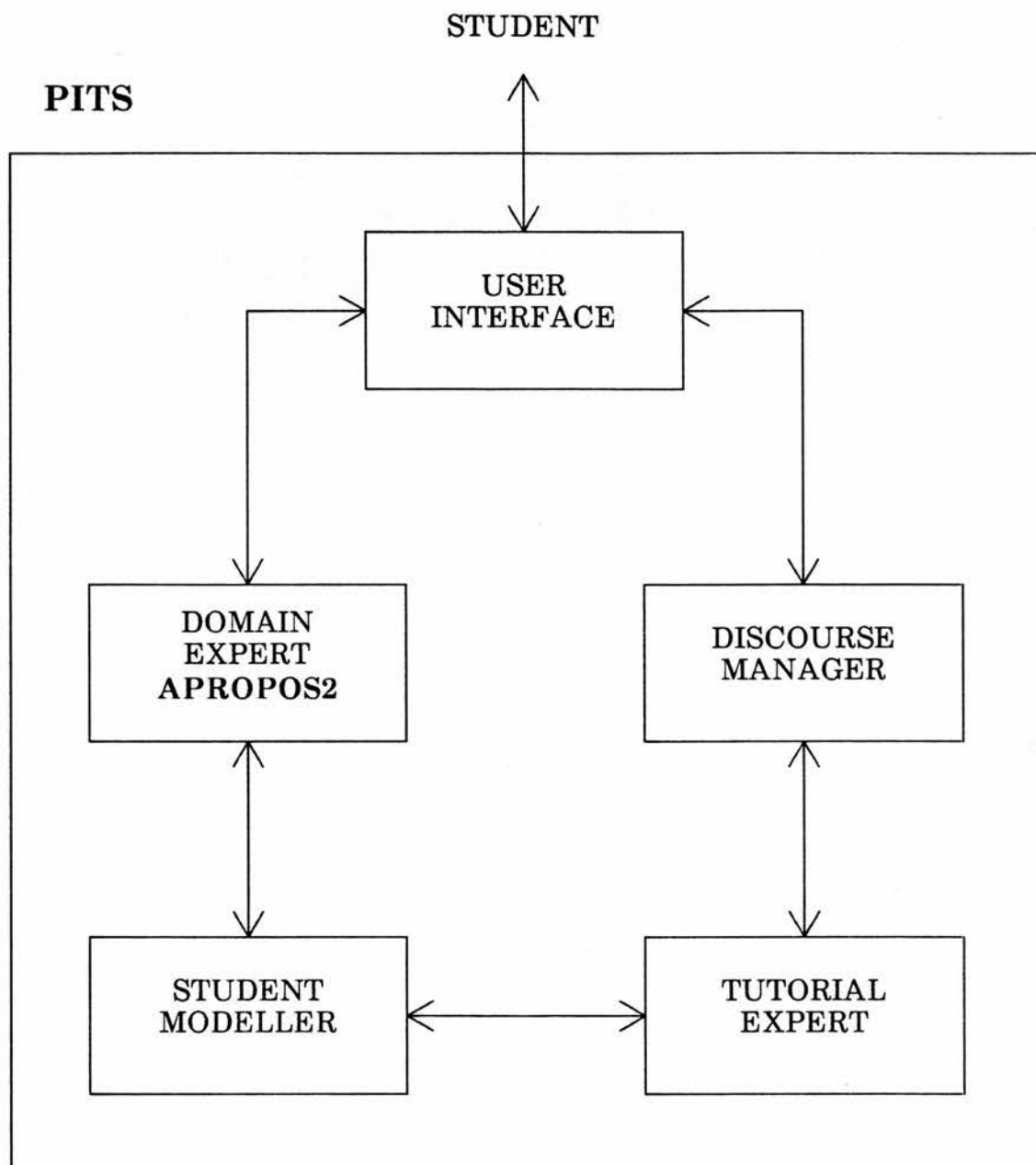


Figure 9-1 : Role of APROPOS2 in a PITS

may be in the form of canned text, guided example, Socratic methods, counter-example or others. The discourse manager decides how to structure the information to present to the student. The student model can be maintained from different information sources such as those suggested by Clancey [Clancey 82] which includes the student's problem-solving behaviour and performance progress, direct questions asked of the student, assumptions based on the student's learning experience, and assumptions based on some measures of the difficulty of the subject material.

9.3 Future Work for a PITS

We have argued in this thesis that multiple sources of program analysis are needed for Prolog program debugging and they have to be interlocked in many ways for greater power. Our objective in the long term is to build a system in which various analysis experts cooperate in the analysis of whether a given student solution meets the specification of the task, and if it does not, provide feedback to the student. A suitable architecture for interlocking these forms of expertise as well as the other components of a complete PITS could be a *blackboard* system. Figure 9-2 shows one view of the design of a PITS incorporating a blackboard architecture. The links in the figure depict the flow of information from the various sources of expertise to the blackboard, and vice versa.

A blackboard architecture combines these forms of expertise to recognise buggy code by coordinating various kinds of observations and deduction. A bug like a mistyping mistake can mislead the subsequent mode analysis; so no one analysis can stand on its own. For dynamic program analysis and student modelling, Ross [Ross 87] proposes the use of an *assumption-based truth maintenance* system [de Kleer 86] to permit different assumptions to be made about different interpretations of a program fragment due to lack of information or certainty. These assumptions can be then sorted out at a later stage of program analysis. For example, if it is unclear at one stage of a program analysis whether a predicate definition argument is intended to be an accumulator argument or a difference-list argument, a PITS can make assumptions about both of these and proceed with the analyses based on each of these assumptions. The assumption-based truth maintenance system is the kind of system which can be incorporated into APROPOS2's best-first search together with some student modelling component. We believe these are worthwhile directions for future research.

In order to integrate all of APROPOS2's analyses to provide apposite critiques, the student modelling and tutoring components are needed which we have not started to work on. The current state of APROPOS2 is that it provides all the analyses it can derive.

We have not started to consider the other components of PITS. Consider the tutorial strategy. The interactive version of APROPOS2 incorporates a first-order model of tutorial

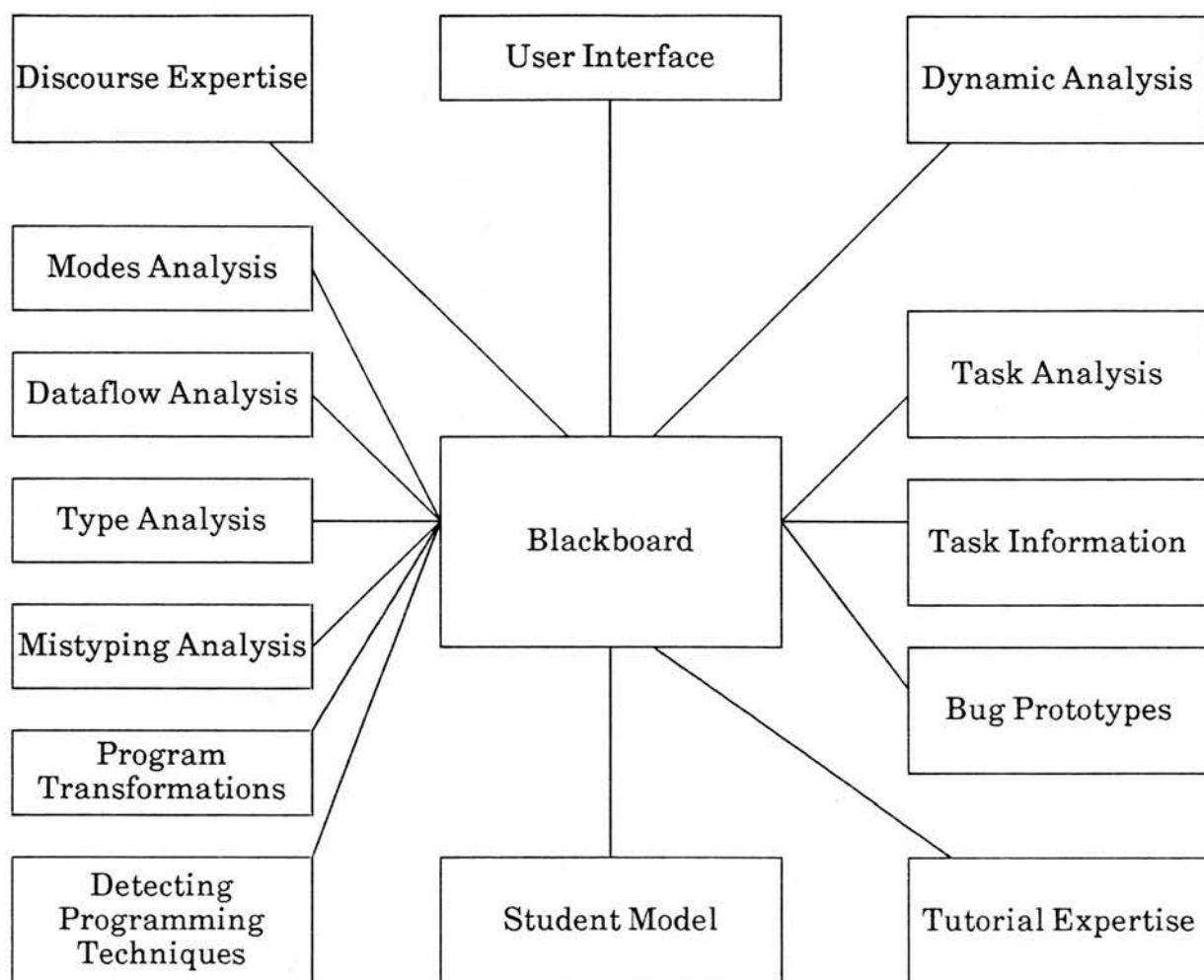


Figure 9-2 : PITS architecture using a blackboard system

intervention. Given a bug, one response suffices and the same response is always given. The response is to describe the bug and the possible misconception (if any) that the bug is highly suggestive of. This is necessarily so since APROPOS2's ability to diagnose programming errors is restricted to the information that is extractable from the buggy programs themselves. APROPOS2 does not discuss misconceptions if there is no strong evidence for them. We believe that this approach is a practical one as the relationship between bugs and misconceptions is far from clear given that not much work has been done to relate Prolog bugs to misconceptions. The research literature reports few systems whose student modelling components allow them to model student misconceptions across his history of progress or even to model it within one problem or task analysis, for instance, to decide whether a bug is just a slip or reflects a deeper misconception. Put another way, our approach is conservative : if APROPOS2 is to be used by itself, then it will attempt to provide enough information to tell the student his bug(s) and the necessary correction(s) to the bug(s), and leaves it to the student himself to infer (and thereby correct) his misconception on his own with the information provided. It is possible to put the student off by providing advice that sometimes postulates misconceptions that the student may not have.

[Ross 87] cites the following example to illustrate the problem of providing an apposite explanation to the student who has written a buggy program. Consider this faulty program to flatten a list :

```
flatten(List, Answer) :-
    flatten(List, [], Answer).           /* line 2 */
flatten([], Answer, Answer) :-
    !.
flatten([Head|Tail], Sofar, Answer) :-   /* line 5 */
    !,
    flatten(Head, Sofar, MoreOfResult),  /* line 7 */
    flatten(Tail, MoreOfResult, Answer). /* line 8 */
flatten(NonListStructure, Sofar, [NonListStructure|Sofar]).
```

What will be the bugs reported by APROPOS? APROPOS would provide at least three program interpretations :

- swap the first arguments of flatten in lines 7 and 8 – the standard terminology for explanation is the accumulator;
- swap the second and third arguments in lines 2 and 9 – the standard terminology for explanation is now 'difference list', since the Answer variable is being progressively instantiated from its head inwards;
- swap the second and third arguments in lines 5, 7 and 8.

It may not be appropriate to give all three critiques since this may confuse the student. For a PITS to choose which critique to comment upon it will have to rely on student modelling – how much knowledge of Prolog has the student shown so far? If he is still working on

accumulator ideas in his efforts, then a PITS should provide him with the first critique as the primary account at least [Ross 87].

To go beyond our current efforts, we have to get into the issue of student modelling and tutorial strategies. Indeed, there are other areas of work which are necessary to the overall effort of building a PITS. For example :

- work to seek better/alternative classification of bugs to form a framework for describing and detecting bugs. Some work has been done by Brna in this area [Brna et al 87].
- work to relate bugs to misconceptions, studies of misconceptions students may have when learning Prolog. Some work has been done by van Someren [van Someren et al 87] who has gathered listings of observed bugs and attempted to relate them to misconceptions.
- study of how human tutors intervene in response to different types of bugs they see in students' programs and a theory of their tutorial intervention. Some work has been done for Pascal by researchers in Yale University [Littman et al 86] in which they observe similar intervention strategies from different tutors in response to the same program and bug.

Chapter 10. Conclusion

In this last chapter, we provide concluding remarks and some thoughts on the generality of APROPOS2's debugging approach for other programming languages. This thesis proposes a program analysis approach for debugging novice Prolog programs in a PITS. We address a small range of list and number manipulation tasks typically encountered in a Prolog programming curriculum. We show how an automated debugger incorporating this approach can analyse students' programs for these tasks to find out if they meet the task specifications, diagnose the bugs in them, and propose the corrections necessary to fix them.

The key features of the program analysis approach are :

- A heuristic best-first search of the program decomposition space to parse a Prolog program into a hierarchical structure of predicate definitions, clauses, subgoals, arguments and terms. A predicate definition is interpreted as an implementation of an algorithm. Thus, the best-first search seeks to uncover the algorithms used and the implementations of these algorithms in a novice Prolog program. This feature is an analysis-by-synthesis approach which enables APROPOS2 to construct predictions on how a program for a task might be implemented. Constructing this space of possible programs and matching them to the student code proceed top-down from analysis of the main predicate definition down to the different levels of auxiliary predicates called by this main predicate. Our work in this thesis thus vouches for the generality of an analysis-by-synthesis approach to accurate debugging of novice programs which is exemplified by the PROUST system for the Pascal language.
- Heuristic code-matching to suggest reference predicates that would match student predicates. This information is used in a program testing approach to find out if running the student predicates on test data would produce the same results as running the reference predicates does. It can also be used to answer oracle queries when debugging using Shapiro's bug diagnosis routines. This is how we have combined static analysis of Prolog programs with dynamic analysis in order to derive the advantages of both approaches. It would be interesting to explore whether the approach of combining static and dynamic analysis is applicable for a different programming language like LISP. Dynamic analysis plays an important part for Prolog as unlike Pascal and LISP there are fewer system predicates or keywords to help anchor a static analysis.

- The use of multiple sources of expertise to analyse Prolog programs for the purpose of understanding student programs, detecting bugs and suggesting fixes to buggy student programs. Information derived from these sources of analysis are used to compute the measure of match between a student program and a reference program. This serves as the heuristic function in our best-first match. Some of these sources of analysis are artifacts of Prolog and we have argued they are necessary for debugging Prolog. While Prolog mode and type analysis are not applicable for other imperative and functional languages, we believe that other forms of analysis like misspelling checking, program transformations and detecting common program schemas all have a part to play in the debugging programs written in other languages.

A program called APROPOS2 incorporating this approach has been built. APROPOS2 which can be seen as a prototype of :

- (1) a teaching tool that provides practice for Prolog novices in learning to write programs for basic list and number manipulation tasks,
- (2) a tool that can be used to evaluate submitted students' solutions for programming tasks. These solutions can be put into a file and run batch on APROPOS2. It is expected that the Prolog instructor would then spend less time vetting through APROPOS2's analyses to validate them than actually marking the solutions himself.

We tested APROPOS2 on a corpus of students' programs for four programming tasks. We evaluated APROPOS2's performance in algorithm recognition, bug detection and bug correction. APROPOS2 achieved a good performance on these scores. Overall, this supports our argument that our program analysis approach is a realistic start for debugging Prolog programs for tutoring purposes. We also propose further work to extend APROPOS2 and to further test its efficacy in debugging novice Prolog programs.

References

- Adam, A. & Laurent, J-P., *LAURA, A System to Debug Student Programs*, AI Journal, 15, 75-122, 1980.
- Anderson, J.R., Boyle, C.F., Corbett, A. & Lewis, M., *Cognitive Modelling and Intelligent Tutoring*, Advanced Computer Tutoring Project, Carnegie-Mellon University, 1987.
- Barr, A., Beard, M. & Atkinson, R., *The Computer as a Tutorial Laboratory*, International Journal of Man-Machine Studies, 8, 567-596, 1976.
- du Boulay, B. & Sothcott, C., *Computers Teaching Programming*, in R.W. Lawler & M. Yazdani (eds), AI and Education, Volume One, Ablex Publishing, 1987.
- Bratko, I., *Prolog Programming for Artificial Intelligence*, Addison-Wesley, 1986.
- Brna, P., *Some Sample Analyses of Prolog Program Bugs*, Blue Book Note 311, Department of AI, University of Edinburgh, 1986a.
- Brna, P., *Possible Prolog Code Errors*, Blue Book Note 325, Department of AI, University of Edinburgh, 1986b.
- Brna, P., Bundy, A., Pain, H. & Lynch, L., *Programming Tools for Prolog Environments*, Advances in Artificial Intelligence, (eds : J. Hallam & C. Mellish), John Wiley, 1987.
- Brna, P. & Pain, H., *Observations of Prolog Novices*, Blue Book Note 259, Department of AI, University of Edinburgh, 1985.
- Brough, D.R. & Hogger, C.J., *The Treatment of Loops in Logic Programming*, Research Report 86/16, Department of Computing, Imperial College of Science and Technology, London, Sept 1986.
- Bruynooghe, M., *Adding Redundancy to Obtain More Reliable and More Readable Prolog Programs*, Proceedings of the 1st International Conference on Logic Programming, 1981.
- Bruynooghe, M., Demoen, B., Callebaut, A. & Janssens, G., *Abstract Interpretation at Work*, Katholieke Universiteit Leuven & BIM, 1986.
- Bundy, A., *Prolog Prototypes, Templates, Schemata and Techniques*, Blue Book Note 296, Department of AI, University of Edinburgh, 1986.
- Bundy, A. et al, *Prolog Programming Techniques*, draft paper, Department of AI, University of Edinburgh, 1987.
- Burton, R.R., *Diagnosing bugs in a simple procedural skill*, in D. Sleeman & J.S. Brown (eds), Intelligent Tutoring Systems, New York, Academic Press, 1982.
- Carver, S.M. & Klahr, D., *Children's Acquisition of Debugging Skills in a LOGO Environment*, Journal of Educational Computing Research, in press.
- Cheatham, T.E., Holloway, G.H. & Townley, J.A., *Symbolic Evaluation and the Analysis of Programs*, IEEE Transactions on Software Engineering, SE-5, July 1979.
- Clancey, W.J., *Tutoring Rules for Guiding a Case Method Dialogue*, in D. Sleeman & J.S. Brown (eds), Intelligent Tutoring Systems, New York, Academic Press, 1982.
- Clark, K.L. & Tarnlund, S-A., *A First Order Theory of Data and Programs*, in Information Processing 77 (B. Gilchrist Ed), pp. 939-944, 1977.

Coombs, M. J. & Stell, J. G., *A Model for Debugging Prolog by Symbolic Execution : The Separation of Specification and Procedure*, Research Report MMIGR137, Department of Computer Science, University of Strathclyde, 1985.

Coombs, M. J., Hartley, R.T. & Stell, J. G., *Debugging User Conceptions of Interpretation Processes*, in Proceedings of AAAI-86, p. 303-307, 1986.

Corbett, A., *personal communication*, 1987.

Cousot, P. & Cousot, R., *A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*, Conference Proceedings of the 4th ACM Symposium on Principles of Programming Languages, 1977, 238-252.

Covington, M.A., *Eliminating Unwanted Loops in Prolog*, ACM SIGPLAN Notices, Vol. 20 No.1, Jan 1985.

Debray, S.K., *Automatic Mode Inference for Prolog Programs*, Technical Report #85/019, Department of Computer Science, State University of New York at Stony Brook, June 1985.

Debray, S.K., *Dataflow Analysis of Logic Programs*, Department of Computer Science, State University of New York at Stony Brook, 1986.

Debray, S.K. & Warren, D.S., *Automatic Mode Inference for Prolog Programs*, Proc 1986 Logic Programming Symposium, Salt Lake City, IEEE Society Press, pp.78-88, 1986.

Dershowitz, N. & Lee, Y-J., *Deductive Debugging*, IEEE Symposium on Logic Programming, 1987.

Doyle, J., *A Truth Maintenance System*, AI-Journal, 12, 231-272, 1979.

Drabent, W. & Maluszynski, J., *Proving Run-Time Properties of Logic Programs*, Department of Computer and Information Science, Linköping University, LITH-IDA-R-86-23, 1986.

Eisenstadt, M., *Retrospective Zooming : A Knowledge-Based Tracing and Debugging Methodology for Logic Programming*, IJCAI 1985.

Eisenstadt, M. & Brayshaw, M., *An Integrated Textbook, Video, and Software Environment for Novice and Expert Prolog Programmers*, Human Cognition Research Laboratory, Open University, Technical Report No. 23, 1987.

Edman, A. & Tarnlund, S-A., *Mechanisation of an Oracle in a Debugging System*, IJCAI 1983, pp. 553-555.

Ehrlich, K. & Soloway, E., *An Empirical Investigation of the Tacit Plan Knowledge in Programming*, in Human Factors in Computer Systems, Ablex, 1983.

Ferrand, G., *Error Diagnosis in Logic Programming : An Adaption of EY Shapiro's Method*, Rapport de Recherche 375, INRIA, 1985.

Fosdick, L.D. & Osterweil, L.J., *Data Flow Analysis in Software Reliability*, in Computing Surveys 8, 3 (1976) 305-330.

Hansson, A. & Tarnlund, S-A., *A Natural Programming Calculus*, Proc. of 6th International Joint Conference on Artificial Intelligence, pp. 348-355, 1979.

Hart, H. E., Nilsson, N. J. & Raphael, B., *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*, IEEE Transactions on SSC, Vol. 4, 1968.

Hart, H. E., Nilsson, N. J. & Raphael, B., *Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths"*, SIGART Newsletter, Vol. 37, 1972.

Huang, X., *Finding Language Errors and Program Equivalence in Automated Programming Advisor*, Research Report 87-13, Department of Computational Science, University of Saskatchewan, 1987.

Hutchings, A.M.J. et al, *Edinburgh Prolog, The New Implementation (NIP) User's Manual*, AI Applications Institute, University of Edinburgh, 1986.

Goldstein, I., *Understanding Simple Picture Programs*, MIT AI Laboratory, TR-405, 1974.

Johnson, W.L., *Intention-Based Diagnosis of Errors in Novice Programs*, PhD Thesis, Yale University, 1985.

Joni, S., Soloway, E., Goldman, R. & Ehrlich, K., *Just So Stories : How the Program Got That Bug*, in Proceedings of the SIGCUE/SIGCAS Symposium on Computer Literacy, 1983.

Kanamori, T. & Horiuchi, K., *Type Inference in Prolog and its Applications*, ICOT Research Center TR-095, 1984.

Kanamori, T. & Seki, H., *Verification of Prolog Programs using an Extension of Execution*, ICOT Research Center TR-096, 1984.

Katz, S. & Manna, Z., *Logical Analysis of Programs*, Communications of the ACM, 19, 188-206, 1976.

O'Keefe, R., *Detecting Non-Terminating Predicates at Compile Time*, Code Note 8, Department of AI, University of Edinburgh, 1981.

Kessler, C.M. & Anderson, J.R., *A Model of Novice Debugging in LISP*, in E. Soloway & S. Iyengar (eds), *Empirical Studies of Programmers*, Ablex Publishing Corporation, 1986.

de Kleer, J., *An Assumption-Based Truth Maintenance System*, AI Journal 28, pp 127-162, 1986.

Kluzniak, F., *Type Synthesis for Ground Prolog*, Proceedings of the Fourth International Conference on Logic Programming, pp 788-816, 1988.

Lever, J., *A Theory of Modes for a Logic Programming Environment*, Department of Computing, Imperial College, London, 1986.

Littman, D., Pinto, J. & Soloway, E., *An Analysis of Tutorial Reasoning about Programming Bugs*, Proceedings of AAAI, 1986.

Lloyd, J.W., *Declarative Error Diagnosis*, Department of Computer Science, University of Melbourne, Technical Report 86/3, 1986.

Looi, C.K. & Ross, P., *Automatic Program Debugging for a Prolog Intelligent Tutoring System*, Department of AI, University of Edinburgh, Research Paper 307, 1987.

Lukey, F.J., *Understanding and Debugging Programs*, International Journal of the Man-Machine Studies, 12, 189-202, 1980.

Lynch, L., *A Thesis Proposal for A Computer Program to Teach Prolog*, Department of AI, Discussion Paper 10, University of Edinburgh, 1986a.

Lynch, L., *Finding Bugs in Prolog Code*, Department of AI, Blue Note 308, University of Edinburgh, 1986b.

Maler O., Scherz, Z. & Shapiro, E., *A New Approach for Introducing Prolog to Naive Users*, Proceedings of the Third International Conference on Logic Programming, 1986.

McCalla, G.I., Bunt, R.B. & Harms, J.J., *The design of the SCENT automated advisor*, Computational Intelligence, Vol. 2, 2, May 1986.

Mellish, C.S., *The Automatic Generation of Mode Declarations for Prolog Programs*, Department of Artificial Intelligence, University of Edinburgh, Research Paper No. 163, 1981.

Mellish, C.S., *Some Global Optimizations for a Prolog Compiler*, in *Journal of Logic Programming*, 1:43-66, 1985.

Miller, M.L., *A Structured Planning and Debugging Environment for Elementary Programming*, *International Journal of the Man-Machine Studies*, 11, 79-95, 1979.

Mishra, P., *Towards a Theory of Types in Prolog*, *Proc. 1984 Symposium on Logic Programming*, Proc. 1984 Symposium on Logic Programming, Atlantic City, IEEE Society Press, pp. 187-196, 1984.

Murray, W.R., *Automatic Program Debugging for Intelligent Tutoring Systems*, AI Laboratory, University of Texas at Austin, TR 86-27, 1986.

Mycroft, R. & O'Keefe, R.A., *A Polymorphic Type System for Prolog*, *Artificial Intelligence*, Vol 23, pp.289-298, 1984.

Nilsson, N., *Principles of AI*, Tioga, Palo Alto, California, 1980.

Pereira, L.M., *Rational Debugging in Logic Programming*, *Proceedings of the Third Logic Programming Conference*, p. 203-204, 1986.

Plummer, D., *SODA : Screen Oriented Debugging Aid*, Software Report 3, Department of Artificial Intelligence, University of Edinburgh, 1986.

Reddy, U.S., *Transformation of Logic Programs into Functional Programs*, *Proc. 1984 Symposium on Logic Programming*, Atlantic City, IEEE Society Press, pp. 187-196, 1984.

Reiser, B., Anderson, J. R., & Farrell, R. G., *Dynamic Student Modelling in an Intelligent Tutor for Lisp Programming*, in *Proceedings of the International Joint Conference on Artificial Intelligence*, 1985.

Rich, C., *Inspection Methods in Programming*, Technical Report AI-TR-604, MIT, 1981.

Ross, P., *Some Thoughts on the Design of an Intelligent Teaching System for Prolog*, AISB Quarterly Summer 1987.

Ross, P., *personal communication*, 1987.

Ruth, G.R., *Intelligent Program Analysis*, *AI Journal*, 7, 65-85, 1976.

Sack, W., et al, *Empirical Evaluation of the Educational Effectiveness of PROUST*, Department of Computer Science, Cognition and Programming Project, Yale University, 1976.

Sack, W., *personal communication*, 1987.

Sedgewick, R., *Algorithms*, Addison-Wesley, 1983.

Shapiro, D., *Sniffer : A System that Understands Bugs*, MIT AI Memo 638, 1981.

Shapiro, E., *Algorithmic Program Debugging*, MIT Press, 1983.

Smolka, G., *Making Control and Data Flow in Logic Programs Explicit*, *Proc. 1984 Symposium on LISP and Functional Programming*, Austin, 1984.

Soloway, E., *From Problems to Programs via Plans : The Content and Structure of Knowledge for Introductory LISP Programming*, *Journal of Educational Computing Research*, Vol. 1(2), 1985.

Soloway, E., *Learning to Program = Learning to Construct Mechanisms and Explanations*, in Communications of the ACM, Volume 29 Number 9, September 1986.

Soloway, E., Rubin, E., Woolf, B., Bonar, J. & Johnson, W., *MENO-II : An AI-Based Programming Tutor*, IEEE Transactions of Software Engineering SE-10, 5, 1984.

Soloway, E., & Ehrlich, K., *Empirical Investigations of Programming Knowledge*, IEEE Transactions of Software Engineering SE-10, 5, 1984.

Soloway, E., Ehrlich, K., Bonar, J. & Greenspan, J., *What do Novices Know about Programming*, in A. Badre & B. Shneiderman (eds), *Directions in Human-Computer Interactions*, Ablex Inc, Norwood, New Jersey, 1982.

Soloway, E., Bonar, J. & Ehrlich, K., *Cognitive Strategies and Looping Constructs : An Empirical Study*, Communications of the ACM 26, November 1983.

van Someren, M. W., *Misconceptions of Beginning Prolog Programmers*, Memorandum 30, Department of Experimental Psychology, University of Amsterdam, 1984.

van Someren, M. W., *Beginning Problems in Learning Prolog*, Memorandum 54, Department of Experimental Psychology, University of Amsterdam, 1985.

van Someren, M. W., Wielinga, B. & Winkels, R., *What's Wrong? Understanding Beginner's Problems with Prolog*, Department of Social Science Informatics & Department of Experimental Psychology, University of Amsterdam, 1987.

Spohrer, J.C. & Soloway, E., *Are the Folk Wisdoms Correct?*, CACM, Vol 29 No 7, pp 624-632, July 1986.

Sterling, L. & Shapiro, E., *The Art of Prolog, Advanced Programming Techniques*, The MIT Press, 1986.

Tarnlund, S-A., *Logic Programming Language Based on A Natural Deduction System*, UPMail Technical Report, No. 6, 1981.

Takahashi, H. & Shibayama, E., *PRESET - A Debugging Environment for Prolog*, Lecture Notes in Computer Science 221, Logic Programming '85, Proceedings of the 4th Conf, Tokyo, Japan, 1985.

Taylor, J. & du Boulay, J.B.H., *Why Novices may find learning Prolog hard*, in J. Rutkowska & C. Crook (eds), *The Child and the Computer: Issues for Deveopmental Psychology*, John Wiley, 1986.

Wagner, H. & Seidel, R., *Program Evaluation*, in H. Wagner & R. Seidel (eds), *Learning Strategies*, 1978.

Warren, D. H. D., Pereira, F. and Pereira, L. M., *PROLOG : the language and its implementation compared with LISP*, SIGPLAN Notices 12 8 (1977), 109-115.

Waters, R.C., *KBEmacs : A Step Toward the Programmer's Apprentice*, MIT AI Laboratory Technical Report 753, 1985.

Wenger, E., *AI and Tutoring Systems : Computational and Cognitive Approaches to the Communication of Knowledge*, Morgan Kaufmann, 1986.

Wertz, H., *Stereotyped Program Debugging : An Aid for Novice Programmers*, International Journal of the Man-Machine Studies, 16, 379-392, 1982.

Wertz, H., *Automatic Correction and Improvement of Programs*, Ellis Horwood, 1987.

Yardeni, E. & Shapiro, E., *A Type System for Logic Programs*, Department of Computer Science, The Weizmann Institute of Science, March 1987.

Zobel, J., *Derivation of Polymorphic Types for Prolog*, Proceedings of the Fourth International Conference on Logic Programming, pp 817-838, 1988.

Appendix I.1 Brief Descriptions of Algorithms for reverse/2

Task Statement:

Write a Prolog program reverse/2 which takes as a list as input and reverses the elements of the list. A goal call "reverse(X, Y)" should succeed with Y instantiated to the reverse of the list X.

Task : reverse(+, -)

First argument: Input list

Second argument: Result

Algorithm 1 : Naive Version

Description : Reverse the tail of the list and then splice the head of the list to the end of the reversed tail to get the answer.

Typical Program :

```
reverse([], []).
reverse([H|T], Res) :-
    reverse(T, Sofar),
    append(Sofar, [H], Res).
```

Algorithm 2 : Inverse Naive Version

Description : Reverse the list without its last element and then splice the last element back to the front of this result to get the answer.

Typical Program :

```
reverse([], []).
reverse(List, [Last|Revlist]) :-
    append(Restlist, [Last], List),
    reverse(Restlist, Revlist).
```

Algorithm 3 : Railway-Shunt Version using an accumulator

Description : Peel off elements of the list from the front successively and use an accumulator to collect these elements in the desired reverse order.

Typical Program :

```
reverse(X, Y) :-
    reverse(X, [], Y).
reverse([], L, L).
reverse([A|X], Y, Z) :-
    reverse(X, [A|Y], Z).
```

or (using Difference Lists)

```
reverse(X, Y) :-
    reverse(X, Y-[]).
reverse([X|Xs], Ys-Zs) :-
    reverse(Xs, Ys-[X|Zs]).
reverse([], Xs-Xs).
```

Task : append/3

Algorithm 1 : Usual Version

Typical Program :

```
append([], L, L).
append([A|X], Y, [A|Z]) :-
    append(X, Y, Z).
```

Appendix I.2 Brief Descriptions of Algorithms for replace_elements/4

Task Statement :

Write a Prolog program `replace_elements/4` which replaces all occurrences of a given element in a list by another and instantiates a given variable to the answer. The arguments should be, in order :

- 1) the element to be replaced
- 2) what to replace it with
- 3) the given list
- 4) a variable to be instantiated to the answer.

Your predicate should not bother to delve inside lists within lists.

Task : `replace_elements(+, +, +, -)`

First argument : Element to be replaced

Second argument : Element to be replaced with

Third argument : Input List

Fourth argument : Result

Algorithm 1 : Procedural Version 1

Description : Check if head of list is identical to element to be replaced. If so, replace it by element to be replaced with and recursively work on the tail of the list.

Typical Program :

```
replace_elements(_, _, [], []).
replace_elements(Old, New, [Old|T], [New|Sofar]) :-
    !,
    replace_elements(Old, New, T, Sofar).
replace_elements(Old, New, [H|T], [H|Sofar]) :-
    replace_elements(Old, New, T, Sofar).
```

Clause Ordering Constraint : The above second clause must come before the third clause.

Algorithm 2 : Declarative Version

Description : Declarative version of Algorithm 1

Typical Program :

```
replace_elements(_, _, [], []).
replace_elements(Old, New, [Old|T], [New|Sofar]) :-
    replace_elements(Old, New, T, Sofar).
replace_elements(Old, New, [H|T], [H|Sofar]) :-
    Old \= H,
    replace_elements(Old, New, T, Sofar).
```

Algorithm 3 : Unfolded Version

Description : Call an auxiliary predicate for the job of replacing each element.

Typical Program :

```
replace_elements(_, _, [], []).
replace_elements(Old, New, [Inp|T], [Out|Sofar]) :-
    replace_one(Old, New, Inp, Out),
    replace_elements(Old, New, T, Sofar).
```

Task : `replace_one/4`

Algorithm 1 : Procedural Version 1

Typical Program :

```
replace_one(Old, New, Old, New) :-
    !.
replace_one(Old, New, H, H).
```

Clause Ordering Constraint : The above first clause must come before the second clause.

Algorithm 2 : Declarative Version

Typical Program :

```
replace_one(Old, New, Old, New).  
replace_one(Old, New, H, H) :-  
    Old \= H.
```

Appendix I.3 Brief Descriptions of Algorithms for atom_count/2

Task Statement:

Write a Prolog program atom_count/2 which accepts a list as first argument and instantiates the variable given as second argument to the number of atoms in the list. The built-in predicate atom/1 succeeds if and only if its argument is an atom.

Example:

```
:- atom_count([a, a, [b,a], []], Res).
```

Res = 3

% ie. the first two "a"'s and the [] (which is an atom, as well as a list).

Task: atom_count(+, -)

First argument: Input list

Second argument: Result

Algorithm 1: Procedural Prefix Version

Description: If non-empty list, check if head of list is an atom and then recursively find the atom_count of the tail of the list. If head of list is an atom, add 1 to the atom_count so far.

Typical Program:

```
atom_count([], 0).
atom_count([H|T], Count) :-
    atom(H),
    !,
    atom_count(T, Sofar),
    succ(Sofar, Count).
atom_count([_|T], Count) :-
    atom_count(T, Count).
```

Clause Ordering Constraint: The above second clause must come before the third clause.

Algorithm 2: Procedural Suffix Version

Description: If non-empty list, recursively find the atom_count of the tail of the list and then check if head of list is an atom. If so, add 1 to the atom_count so far.

Typical Program:

```
atom_count([], 0).
atom_count([H|T], Count) :-
    atom_count(T, Sofar),
    atom(H),
    !,
    succ(Sofar, Count).
atom_count([_|T], Count) :-
    atom_count(T, Count).
```

Clause Ordering Constraint: The above second clause must come before the third clause.

Algorithm 3: Declarative Prefix Version

Description: Declarative version of Algorithm 1

Typical Program:

```
atom_count([], 0).
atom_count([H|T], Count) :-
    atom(H),
    atom_count(T, Sofar),
    succ(Sofar, Count).
atom_count([H|T], Count) :-
    \+ atom(H),
    atom_count(T, Count).
```

Algorithm 4 : Declarative Suffix Version

Description : Declarative version of Algorithm 2

Typical Program :

```
atom_count([], 0).  
atom_count([H|T], Count) :-  
    atom_count(T, Sofar),  
    atom(H),  
    succ(Sofar, Count).  
atom_count([H|T], Count) :-  
    atom_count(T, Count),  
    \+ atom(H).
```

Appendix I.4 Brief Descriptions of Algorithms for fringe_length/2

Task Statement :

Write a Prolog program `fringe_length/2` which expects a list as its first argument, and which instantiates the variable given as its second argument to the number of elements in the list, or in any lists in the lists, or in any lists in ... in the lists. If an element appears more than once, it counts more than once. A list does not count as an element : instead it contributes its `fringe_length` (which is 0 for an empty list) to the total. Examples :

```
:- fringe_length([a, a, [b, a]], Res).
```

Res = 4

% ie. a, a, b and a

```
:- fringe_length([a, [[a], []], [b,c]], Res).
```

Res = 4

% ie. a, a, b and c

```
:- fringe_length([[], [], []], Res).
```

Res = 0

Task : `fringe_length(+, -)`

First argument : Input list

Second argument : Result

Algorithm 1 : Procedural Version 1

Description : Check if head of the list is the empty list. If so, `fringe_length` is the `fringe_length` of the tail of list. If not, check if head of list is an atom. If so, add 1 to `fringe_length` of tail of list for the result. If not, `fringe_length` is the sum of `fringe_lengths` of head and tail of list.

Typical Program :

```
fringe_length([], 0).
fringe_length([_|T], Count) :-
    !,
    fringe_length(T, Count).
fringe_length([H|T], Count) :-
    atom(H),
    !,
    fringe_length(T, Sofar),
    succ(Sofar, Count).
fringe_length([H|T], Count) :-
    fringe_length(H, I),
    fringe_length(T, J),
    Count is I + J.
```

Clause Ordering Constraint : The above second clause must come before the third clause, the third clause before the fourth clause.

Algorithm 2 : Procedural Version 2

Description : Check if head of the list is an atom and not the empty list. If so, add 1 to fringe_length of tail of list for the result. If not, fringe_length is the sum of fringe_lengths of head and tail of list.

Typical Program :

```
fringe_length([], 0).
fringe_length([H|T], Count) :-
    atom(H),
    H \= [],
    !,
    fringe_length(T, Sofar),
    succ(Sofar, Count).
fringe_length([H|T], Count) :-
    fringe_length(H, I),
    fringe_length(T, J),
    Count is I + J.
```

Clause Ordering Constraint : The above second clause must come before the third clause.

Algorithm 3 : Declarative Version 1

Description : Declarative version of Algorithm 1

Typical Program :

```
fringe_length([], 0).
fringe_length([_|T], Count) :-
    fringe_length(T, Count).
fringe_length([H|T], Count) :-
    atom(H),
    H \= [],
    fringe_length(T, Sofar),
    succ(Sofar, Count).
fringe_length([H|T], Count) :-
    \+ atom(H),
    fringe_length(H, I),
    fringe_length(T, J),
    Count is I + J.
```

Algorithm 4 : Declarative Version 2

Description : Declarative version of Algorithm 2

Typical Program :

```
fringe_length([], 0).
fringe_length([H|T], Count) :-
    atom(H),
    H \= [],
    fringe_length(T, Sofar),
    succ(Sofar, Count).
fringe_length([H|T], Count) :-
    \+ atom(H),
    fringe_length(H, I),
    fringe_length(T, J),
    Count is I + J.
```

Algorithm 5 : Declarative Version 3

Description : Check if first argument is an atom. If so, add 1 to fringe_length so far for the result else call fringe_length/2 on head and tail of list recursively and add up their fringe_lengths to get the result.

Typical Program :

```
fringe_length([], 0).
fringe_length(A, 1) :-
    atom(A).
fringe_length([H|T], Count) :-
    fringe_length(H, I),
    fringe_length(T, J),
    Count is I + J.
```

(The next algorithm has not yet been incorporated into APROPOS2.)

Algorithm 6 : Flatten-and-Count Version

Description : Check if head of the list is an atom and not the empty list. If so, add 1 to fringe_length of tail of list for the result. If not (head is a sublist then), append the head of the list to the tail of the list to get Flat and fringe_length is the fringe_length of Flat.

Typical Program :

```
fringe_length([], 0).
fringe_length([H|T], Count) :-
    (\+ atom(H) ; atom = []),
    append(H, T, Flat),
    fringe_length(Flat, Count).
fringe_length([H|T], Count) :-
    atom(H),
    \= [],
    fringe_length(T, I),
    Count is I + 1.
```

Appendix I.5 Brief Descriptions of Algorithms for sorting/2

Task Statement :

Write a Prolog program `sorting/2` which takes a list of numbers as input and sorts the numbers in the list in ascending order. A goal call "`sorting(X, Y)`" where X is the input list should succeed with Y instantiated to the sorted list. Use the predicate name `sorting` instead of `sort` which is an Edinburgh Prolog built-in predicate.

Task : `sorting(+, -)`

First argument : Input list of numbers

Second argument : Result

Algorithm 1 : Bubble Sort

Description : Swap two elements in the input list in the correct order, and then recursively sort again with the new list.

Typical Program :

```
sorting(List, Res) :-  
    swap(List, SwappedList),  
    !,  
    sorting(SwappedList, Res).  
sorting(Sorted, Sorted).
```

Clause Ordering Constraint : The above first clause must come before the second clause.

{ There are many nontrivial implementation variants, for example, the first clause may be written as :

```
sorting(List, Res) :-  
    append(X, [A, B/Y], List),  
    A > B,  
    append(X, [B, A/Y], SwappedList),  
    sorting(SwappedList, Res).
```

and the second clause as :

```
sorting(List, List) :-  
    ordered(List). }
```

Algorithm 2 : Insertion Sort

Description : First sort the tail of the input list, and then insert the first element back into the proper place in the sorted tail.

Typical Program :

```
sorting([Head|Tail], Res) :-  
    sorting(Tail, SortedTail),  
    insert(Head, SortedTail, Res).  
sorting([], []).
```

Algorithm 3 : Permutation Sort

Description : Permute the input list, and test if the permuted list is sorted.

Typical Program :

```
sorting(List, Res) :-  
    permutation(List, PermutedList),  
    ordered(PermutedList, Res).
```

Algorithm 4 : Quick Sort

Description : Divide the input list into the elements smaller than the first element and the elements larger than the first element, and then recursively sort these two pieces, and finally compose the result from the smaller elements, followed by the first element, and then the larger elements.

Typical Program :

```
sorting([Head|Tail], Res) :-  
    partition(Tail, Head, Littles, Bigs),  
    sorting(Littles, SortedLittles),  
    sorting(Bigs, SortedBigs),  
    append(SortedLittles, [Head|SortedBigs], Res).  
sorting([], []).
```

Many auxiliary predicates are used, but their algorithms are not described here.

Appendix II. A Session Listing of APROPOS1's Analyses of a Student's reverse/2 programs

/* APROPOS1's output in normal print, student's input in **bold** print, and added annotations in *italics* print */

: ? - **apropos.**

Programming Assignment : LIST-REVERSAL

Write a Prolog program reverse/2 which takes as a list as input and reverses the elements of the list. A goal call "reverse(X, Y)" should succeed with Y instantiated to the reverse of the list X.

For example,

 ?- reverse([1,2,3], Y).

should succeed with Y instantiated to [3,2,1].

 ?- reverse([1,2,3], [3,2,1]).

should succeed.

Now, use another window to type in your program in a file in the your current directory, and give it a filename.

What is the name of your program file?

|: **newone**

Now, analysing your Prolog source code ...

Your program clauses are :

```
reverse([], []).
reverse([H|T], Y) :-
    append(H, Y, Y),
    reverse(T, Y).
```

Well, in this clause :

```
reverse([H|T], Y) :-
    append(H, Y, Y),
    reverse(T, Y).
```

reverse/2 calls append/3 which is not defined.

Your base case :

```
reverse([], []).
```

seems fine.

Let's run your program on an example : reverse([], X).

Your program produces solution : reverse([], []) which is correct, and no more solutions.

Analysing clause :

```
reverse([H|T], Y) :-
    append(H, Y, Y),
    reverse(T, Y).
```

Argument 2 is the reverse of argument 1. Well, [H|T] is a longer list than T, so the reverse of [H|T] cannot be the same as that of T.

Let's run your program on an example : reverse([a,b], X).

Error : missing solution reverse([a,b], [b,a]).

{ APROPOS1's analysis and comments at this point are insufficient. As it had discovered a violation of a well-founded ordering relation, it did not proceed to analyse the append(H, Y, Y) subgoal. APROPOS2, on the other hand, would comment on this append subgoal, as it has to explain all code discrepancy. }

Do you want to edit your file? yes

{ Student edited his program and APROPOS1 resumes its analysis. }

Your program clauses are :

```
reverse([], Y) :-  
reverse([H|T], Y) :-  
    reverse(T, [H|Y]).
```

Your base case :

```
reverse([], Y).
```

the reverse of [] is [] and not Y.

Let's run your program on an example : reverse([], X).

Your program produces solution : reverse([], Y) which is too general, and no more solutions.

Analysing clause :

```
reverse([H|T], Y) :-  
    reverse(T, [H|Y]).
```

Argument 2 is the reverse of argument 1. Well, [H|T] is a longer list than T, so the reverse of [H|T] cannot be shorter in length than that of T.

In the second argument, Y in the head goal is accumulated to [H|Y] in the recursive subgoal. However, this accumulated result is not passed back! Just suppose the subgoal

```
reverse(T, [H|Y])
```

succeeds, then program flow returns to the head goal

```
reverse([H|T], Y)
```

which also succeeds, but the [H|Y] result is lost!

You have to use an extra argument to pass the results all the way back from the base case, or you have to rewrite the clause, so that the result you want is not lost.

Let's run your program on an example : reverse([a,b], X).

Your program produces solution : reverse([a,b], Y) which is too general, and no more solutions.

{ We have not placed much emphasis on phrasing these advice messages into APROPOS. Indeed, they can be better phrased. }

Do you want to edit your program now? yes

{ Student edited his program and APROPOS1 resumes its analysis. }

Your program clauses are :

```
reverse([], [], Y) :-  
reverse([H|T], Y, X) :-  
    reverse(T, [H|Y], X).  
reverse(X, Y) :-  
    reverse(X, [], Y).
```

Well, these clauses :

```
reverse([], [], Y) :-  
reverse([H|T], Y, X) :-  
    reverse(T, [H|Y], X).
```

> > > Dataflow error in argument 2 of reverse/3.

This will give you an incorrect computation as this argument will build up into a larger structure during recursive calls. The recursive subgoal call will never match the base case.

Argument 3 of reverse/3 is used to pass back a result accumulated or obtained when recursion terminates on a base case.

Analysing clause :

```
reverse([H|T], Y, X) :-  
    reverse(T, [H|Y], X).
```

This clause looks fine. Y in the head goal is accumulated to [H|Y] in the recursive goal.

Coming back to your base clause :

```
reverse([], [], Y).
```

It does not seem quite right. Replace it by

```
reverse([], X, X).
```

Let's run your program on an example : reverse([], X).

Your program produces solution reverse([], Y) which is too general, and no more solutions.

Analysing program clause :

```
reverse(X, Y) :-  
    reverse(X, [], Y).
```

Let's run your program on an example : reverse([a,b], X).

Error : missing solution reverse([a,b], [b,a]).

Do you want to edit your program now? **yes**

{ Student edited his program and APROPOS1 resumes its analysis. }

Your program clauses are :

```
reverse([], X, X).
reverse([H|T], Y, X) :-
    reverse(T, [H|Y], X).
reverse(X, Y) :-
    reverse(X, [], Y).
```

Argument 3 of reverse/3 is used to pass back a result accumulated or obtained when recursion terminates on a base case.

Analysing clause :

```
reverse([H|T], Y, X) :-
    reverse(T, [H|Y], X).
```

This clause looks fine. Y in the head goal is accumulated to [H|Y] in the recursive goal.

Coming back to your base clause :

```
reverse([], X, X).
```

It seems fine.

Let's run your program on an example : reverse([], X).

Your program produces solution reverse([], []) which is right, and no more solutions.

Analysing program clause :

```
reverse(X, Y) :-
    reverse(X, [], Y).
```

Let's run your program on an example : reverse([a,b], X).

Your program produces solution reverse([a,b], [b,a]) which is right, and no more solutions.

Let's run your program on an example : reverse([1,2,3], X).

Your program produces solution reverse([1,2,3], [3,2,1]) which is right, and no more solutions.

Do you want to try another reverse/2 program on APROPOS1 again? **no**

*{ Student ends his session at this point and here are some of his comments on APROPOS1 :
"Yes, it founded errors in my program. I think it found them all. It provided very useful
comments on designing this program; it seemed to tell me nearly everything I needed. It is
difficult to understand recursion as an iterative process ... the variables all seem fuzzy in this
declarative language, and etc etc ..." }*

Appendix III. A Session Listing of APROPOS-AS's Analyses of a Student's programs

/* APROPOS-AS's output in normal print, student's input in **bold** print, and added annotations in *italics* print */

: ? - **apropos.**

Programming Assignment : CALCULATING POPULATION-DENSITY

Asserted in the Prolog database are clauses :

% figures given in millions

pop(britain, 56).

pop(india, 685).

... more data ...

% figures given in thousands of square miles

area(britain, 94).

area(india, 1262).

... more data ...

Write a Prolog program density/0 which, for each country C, finds its population P and its area A, and calculates its population density D, and prints C, P, A, D.

For example,

?- density.

should succeed with these results printed out

Country	Population(m)	Area(1000 sq m)	Density
britain	56	94	0.595745
india	685	1262	0.542789
...			

The main task is to get these figures. The format of the output is not important.

Now, use another window to type in your program in a file in the your current directory, and give it a filename.

What is the name of your program file?

|: **density**

Now, analysing your Prolog source code ...

Your program clauses are :

```
density :-  
    writedense('Country', 'Pop', 'Area', 'Density'),  
    nl,  
    density(_),  
    fail,  
    nl.  
density(C) :-  
    pop(C, P),  
    area(C, A),  
    D is P/A,  
    writedense(C, P, A, D).  
writedense(C, P, A, D) :-  
    write(C),  
    put(9),  
    put(9),  
    write(P),  
    put(9),  
    write(A),  
    put(9),  
    write(D),  
    nl.
```

You have a failure-driven loop in the clause for density/0. This clause will eventually fail when the loop ends. If you want the call to the loop to succeed eventually, insert an unit clause after this clause.

One of the goals for this task is to have a clause that makes a call to the failure-driven loop succeed. You do not seem to have achieved this goal.

So, your program seems to be incorrect.

Do you want to edit your file? **yes**

{ Student edited her program and APROPOS-AS resumes its analysis. }

Your program clauses are :

```
density :-  
    dense,  
    nl.  
dense :-  
    writedense('Country', 'Pop', 'Area', 'Density'),  
    nl,  
    dense(_),  
    fail.  
dense(C) :-  
    pop(C, P),  
    area(C, A),  
    D is P/A,  
    writedense(C, P, A, D).  
writedense(C, P, A, D) :-  
    write(C),  
    put(9),  
    put(9),  
    write(P),  
    put(9),  
    write(A),  
    put(9),  
    write(D),  
    nl.
```

Looking at this clause :

```
density :-  
    dense,  
    nl.
```

You have a failure-driven loop in the clause for density/0. This clause will eventually fail when the loop ends. If you want the call to the loop to succeed eventually, insert an unit clause after this clause.

One of the goals for this task is to have a clause that makes a call to the failure-driven loop succeed. You do not seem to have achieved this goal.

So, your program seems to be incorrect.

Do you want to edit your file? **yes**

{ Student edited her program and APROPOS-AS resumes its analysis. }

Your program clauses are :

```
density :-  
    writedense('Country', 'Pop', 'Area', 'Density'),  
    nl,  
    dense(_),  
    fail.  
density :-  
    nl.  
dense(C) :-  
    pop(C, P),  
    area(C, A),  
    D is P/A,  
    writedense(C, P, A, D).  
writedense(C, P, A, D) :-  
    write(C),  
    put(9),  
    put(9),  
    write(P),  
    put(9),  
    write(A),  
    put(9),  
    write(D),  
    nl.
```

There is a clause

```
density :-  
    nl.
```

to succeed when the loop ends.

Your program seems fine.

Let's try the program on the two set of data given above.

|? - density.

Country	Pop	Area	Density
britain	56	94	0.595745
india	685	1262	0.542789

yes

Do you want to try another density/0 program on APROPOS-AS again? **yes**

{ Student intentionally added an error in her program and tries it out again. }

What is the name of your program file?

|: **density**

Now, analysing your Prolog source code ...

Your program clauses are :

```
density :-
    writedense('Country', 'Pop', 'Area', 'Density'),
    nl,
    dense(_),
    fail.
density :-
    nl.
dense(C) :-
    pop(C, P),
    area(C, A),
    D is P + A,
    writedense(C, P, A, D).
writedense(C, P, A, D) :-
    write(C),
    put(9),
    put(9),
    write(P),
    put(9),
    write(A),
    put(9),
    write(D),
    nl.
```

There is a clause

```
density :-
    nl.
```

to succeed when the loop ends.

This is a simple failure-driven loop.

One of the goals for this task is to compute the population density of a country. You do not seem to have achieved this goal.

So, your program seems to be incorrect.

Do you want to edit your file? **no**

*{ Student ends her session at this point and here are some of her comments on APROPOS-AS :
"Well, it picked up the error when I typed '+' rather than '/' - was interested to try (as I did actually do that the first time I edited the file. It would be nice if the user could see the execution of the code under discussion - perhaps call it with the trace running? the bit about "include an unit clause after this clause" had me baffled - it's not clear to me which "this clause" you mean. It wasn't until I thought about how to get the Prolog to work, instead of inserting unit clauses after the call to "fail", and bundling the whole thing up in another clause altogether, that I worked out what it meant." }*



THE UNIVERSITY *of* EDINBURGH

PAGE MISSING IN ORIGINAL

Appendix IV. Examples of APROPOS2's Analyses of Student's reverse/2 programs

Your program clauses are :

```
reverse([], []).
reverse([N], [N]).
reverse([A|B], [C|A]) :-
    reverse(B, C).
```

Your program seems to be incorrect.
Now, this is APROPOS2's critique on your program :

In the clause :
reverse([A|B], [C|A]) :-
 reverse(B, C).

if A in [C|A] is not a list, perhaps an atom, then [C|A] may not be what you intend.
A note about Prolog syntax : [a|[b]] means [a,b] while [a|[b,c]] means [a,b,c] but [a|b] is a different structure from [a,b].

Clause 1 of reverse/2 seems fine.
The reverse of the empty list is the empty list.

Clause 2 of reverse/2 seems to be redundant.

Clause 3 of reverse/2 does not seem quite right.
Replace reverse([A|B], [C|A]) with reverse([A|B], X).
A subgoal like append(C, [A], X) seems to be missing.
This is what your clause should look like :

```
reverse([A|B], X) :-
    reverse(B, C),
    append(C, [A], X).
```

This recursive case says that the reverse of a non-empty list can be found by reversing the tail of the list, and then appending a list consisting only of the first element of the original to the result of reversing the tail.

This predicate definition seems to be missing :
append([H|X], Y, [H|Z]) :-
 append(X, Y, Z).
append([], L, L).

The first clause is the recursive clause.
The second clause says that the result of appending an empty list with the list L is the list L itself.

Analysis of a reverse/2 program using the naive algorithm

Your program clauses are :

```
reverse([], []).
reverse([L, [H|RT]] :-
    append(T, [H], L),
    reverse(T, RT).
```

Your program seems to be incorrect.

Now, this is APROPOS2's critique on your program :

Well, in this clause :

```
reverse([L, [H|RT]] :-
    append(T, [H], L),
    reverse(T, RT).
```

reverse/2 calls append/3 which is not defined.

Clause 1 of reverse/2 seems fine.

The reverse of the empty list is the empty list.

Clause 2 of reverse/2 seems fine.

This recursive case says that the reverse of a non-empty list can be found by reversing the list without the last element, and then appending a list consisting only of the last element of the original with the result of reversing the list without the last element.

This predicate definition seems to be missing :

```
append([H|X], Y, [H|Z]) :-
    append(X, Y, Z).
append([], L, L).
```

The first clause is the recursive clause.

The second clause says that the result of appending an empty list with the list L is the list L itself.

Analysis of a reverse/2 program using the inverse naive algorithm

Your program clauses are :

```
reverse(X, Y) :-  
    reverse(X, [], Y).  
reverse([], X, X).  
reverse([H|T1], [H2|T2], Z) :-  
    reverse(T1, [H2, H1|T1], Z).
```

Your program seems to be incorrect.
Now, this is APROPOS2's critique on your program :

Clause 1 of reverse/3 seems fine.
This clause instantiates the accumulated result to the result variable.

Clause 2 of reverse/3 does not seem quite right.
Replace reverse([H|T1], [H2|T2], Z) with reverse([H1|T1], X, Z).
Replace reverse(T1, [H2, H1|T1], Z) with reverse(T1, [H1|X], Z).
This is what your clause should look like :
reverse([H1|T1], X, Z) :-
 reverse(T1, [H1|X], Z).

This clause uses an argument to accumulate the result as recursion proceeds, and another argument to be instantiated to the accumulated result when recursion ends.

Clause 1 of reverse/2 seems fine.
reverse/2 calls another predicate to do the work.

Analysis of a reverse/2 program using the railway-shunt algorithm

Your program clauses are :

```
reverse([], Y).
reverse([], []).
reverse(A, B) :-
    recur(A, B).
recur([H|T], Y) :-
    recur(T, [H|Y]).
```

Your program seems to be incorrect.
Now, this is APROPOS2's critique on your program :

recur/2 has no terminating proof.
Either there is a missing base case or it calls other predicates that have no terminating proofs themselves.

In the clause :

```
reverse([], Y).
```

Y appears only once. Recall that if variable appears once only, we do not have to invent a name for it. We can use the so-called anonymous variable, which is written as a single underscore character "_".

Clause 1 of recur/2 does not seem quite right.
Replace recur([H|T], Y) with recur([H|T], Y, X)
and recur(T, [H|Y]) with recur(T, [H|Y], X).
This is what your clause should look like :

```
recur([H|T], Y, X) :-
    recur(T, [H|Y], X).
```

This clause uses an argument to accumulate the result as recursion proceeds, and another argument to be instantiated to the accumulated result when recursion stops.

In your predicate definition of recur/2, a clause like :

```
recur([], X, X).
```

seems to be missing.

This clause instantiates the accumulated result to the result variable.

Clause 1 of reverse/2 seems to be redundant.

Clause 2 of reverse/2 seems to be redundant.

Clause 3 of reverse/2 does not seem quite right.

Replace recur(A, B) with recur(A, [], B).

This is what your clause should look like :

```
reverse(A, B) :-
    recur(A, [], B).
```

reverse/2 calls another auxiliary predicate recur/3 to do the job.

Analysis of a reverse/2 program attempting the railway-shunt algorithm

Appendix V. An Independent Judgement of APROPOS2's Analyses of reverse/2 programs

The following form was given to a Prolog lecturer in the Department to seek an independent judgement of APROPOS2's analyses of students' reverse/2 programs :

This study seeks an independent judgement of APROPOS2's analyses of students' reverse/2 programs. In this form, we list students' programs for reverse/2. For each program listed, we summarise APROPOS2's analysis of the program. Your task is to comment on APROPOS2's analysis.

For each program, we list these information :

Frequency :

{ no of such programs in corpus, ignoring a relabelling of variables and inconsequential clause orderings }

Is program correct?

{ does the program work for reverse(+, ?) and produce the correct answer on its first run? }

Is program recognised (by APROPOS2) as correct?

{ did APROPOS2 find the program to be correct? }

Algorithm used :

{ algorithm intended in the student's program in my judgement }

Algorithm detected :

{ algorithm detected by APROPOS2 }

No of bugs in program :

{ no of bugs in the program in my judgement }

No of bugs detected and corrected :

{ no of bugs detected and corrected by APROPOS2 }

No of bugs detected but corrected wrongly :

{ no of bugs detected but corrected wrongly by APROPOS2 }

No of bugs not detected :

{ no of bugs present in program but not detected by APROPOS2 }

No of false alarms :

{ no of bugs not present in program but flagged by APROPOS2 }

Also attached with this form are descriptions of the various algorithms used by APROPOS2 for reverse/2 (as in Appendix I.1), a note summarising APROPOS2's analysis of bugs describing, in particular, how bugs are counted (as we have described in Section 4.1), and the full listing of all program analyses of APROPOS2 (omitted here).

Please go through the list of programs and comment on APROPOS2's judgement of the algorithm used, and APROPOS2's detection and correction of bugs.

Program version 1 :**Frequency : 6**

```
reverse(X, Y) :-  
    reverse(X, [], Y).  
reverse([], L, L).  
reverse([H|T], Y, Z) :-  
    reverse(T, [H|Y], Z).
```

Is program correct? yes**Is program recognised as correct? yes****Algorithm used : railway-shunt****Algorithm detected : railway-shunt****No of bugs in program : 0****No of bugs detected and corrected : 0****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**

Program version 2 :**Frequency : 1**

```
reverse(X, Y) :-  
    reverse(X, -, Y).                % bug - reverse(X, [], Y).  
reverse([], L, L).  
reverse([H|T], Y, Z) :-  
    reverse(T, [H|Y], Z).
```

Is program correct? no**Is program recognised as correct? no****Algorithm used : railway-shunt****Algorithm detected : railway-shunt****No of bugs in program : 1****No of bugs detected and corrected : 1****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**

Program version 3 :**Frequency : 2**

```
reverse([], L, L).  
reverse([], []).  
reverse(X, Y) :-  
    reverse(X, [], Y).  
reverse([H|T], Y, Z) :-  
    reverse(T, [H|Y], Z).
```

Is program correct? yes**Is program recognised as correct? yes****Algorithm used : railway-shunt****Algorithm detected : railway-shunt****No of bugs in program : 0****No of bugs detected and corrected : 0**

{Redundancy of "reverse([], []). " is detected but not counted as a bug.}

No of bugs detected but corrected wrongly : 0**No of bugs not detected : 0****No of false alarms : 0***Comment : This program will produce two solutions for reverse([], X).*

Program version 4 :**Frequency : 3**

```
reverse(X, Y) :-  
    rev(X, [], Y).  
rev([], L, L).  
rev([H|T], Y, Z) :-  
    rev(T, [H|Y], Z).
```

Is program correct? yes**Is program recognised as correct? yes****Algorithm used : railway-shunt****Algorithm detected : railway-shunt****No of bugs in program : 0****No of bugs detected and corrected : 0****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**

Program version 5 :**Frequency : 1**

```
reverse(X, Y) :-  
    reverse(X, [], Y).  
reverse([], L, L).  
reverse([H1|T1], [H2|T2], Z) :-  
    reverse(T, [H2, H1|T1], Z).  
% bug - reverse([H1|T1], X, Z) :-  
% bug - reverse(T, [H1|X], Z).
```

Is program correct? no**Is program recognised as correct? no****Algorithm used : railway-shunt****Algorithm detected : railway-shunt****No of bugs in program : 2****No of bugs detected and corrected : 2****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**

Program version 6 :**Frequency : 1**

```
reverse(List, Tsil) :-  
    reverse(List, [], Tsil).  
reverse([X|List], Temp, [X|Temp]) :-  
    atom(List).  
reverse([X|List], Temp, Temp) :-  
    \+ atom(List),  
    reverrse(List, [X|Temp], [X|Temp]).  
% bug - extra clause  
% bug - reverse([X|List], Temp, Y) :-  
% bug - extra subgoal  
% bug - reverse(List, [X|Temp], Y).  
% bug - missing clause : reverse([], X, X).
```

Is program correct? no**Is program recognised as correct? no****Algorithm used : railway-shunt****Algorithm detected : railway-shunt****No of bugs in program : 5****No of bugs detected and corrected : 5****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**

Program version 7 :**Frequency : 1**

```
reverse(X, Y) :-  
    reverse(X, [], Y).  
reverse([], L, L).  
reverse([H|L], [X|L1], L2) :-  
    reverse(L, L1, L2).  
% bug - reverse([H|L], L1, L2) :-  
% bug - reverse(L, [H|L1], L2).
```

Is program correct? no**Is program recognised as correct?** no**Algorithm used :** railway-shunt**Algorithm detected :** railway-shunt**No of bugs in program :** 2**No of bugs detected and corrected :** 2**No of bugs detected but corrected wrongly :** 0**No of bugs not detected :** 0**No of false alarms :** 0

Program version 8 :**Frequency : 1**

```
reverse(A, B) :-  
    reverse(A, [B]).  
rev([], [B|B]).  
rev([Head|Tail], [B|Temp]) :-  
    reverse(Tail, [B, Head|Temp]).  
% bug - reverse(A, B) :-  
% rev(A, [], B).  
% bug - rev([], X, X).  
% bug - rev([Head|Tail], X, Y) :-  
% bug - rev(Tail, [Head|X], Y).
```

Is program correct? yes**Is program recognised as correct?** no**Algorithm used :** railway-shunt**Algorithm detected :** railway-shunt**No of bugs in program :** 0**No of bugs detected and corrected :** 0**No of bugs detected but corrected wrongly :** 0**No of bugs not detected :** 0**No of false alarms :** 4

Program version 9 :**Frequency : 1**

```
reverse([], []).  
reverse(L, [H|RT]) :-  
    append(T, [H], L),  
    reverse(T, RT).  
% bug - missing append/3 definition
```

Is program correct? no**Is program recognised as correct?** no**Algorithm used :** inverse-naive**Algorithm detected :** inverse-naive**No of bugs in program :** 1**No of bugs detected and corrected :** 1**No of bugs detected but corrected wrongly :** 0**No of bugs not detected :** 0**No of false alarms :** 0

Program version 10 :**Frequency : 1**

```
reverse([], Y).  
reverse(X, Y) :-  
    reverse([H|T], Y).
```

Is program correct? no**Is program recognised as correct? no****Algorithm used : unclear****Algorithm detected : inverse-naive***{results of analysis not shown here since algorithm is unclear}*

Program version 11 :**Frequency : 1**

```
reverse([], []).  
reverse([HX|X], Y) :-  
    append(Y, HX).
```

Is program correct? no**Is program recognised as correct? no****Algorithm used : unclear****Algorithm detected : naive***{results of analysis not shown here since algorithm is unclear}*

Program version 12 :**Frequency : 1**

```
reverse([], []).  
reverse([H1|T1], [H2|T2]) :-  
    reverse(T1, T2)  
    H2 is H1.
```

Is program correct? no**Is program recognised as correct? no****Algorithm used : unclear****Algorithm detected : 2 alternative interpretations – naive & inverse naive***{results of analysis not shown here since algorithm is unclear}*

Program version 13 :**Frequency : 1**

```
reverse([], []).  
reverse([H|T], [H|Y]) :-  
    reverse(T, Y).
```

Is program correct? no**Is program recognised as correct? no****Algorithm used : unclear****Algorithm detected : 2 alternative interpretations – naive & inverse naive***{results of analysis not shown here since algorithm is unclear}*

Program version 14 :**Frequency : 1**

```
reverse([], []).  
reverse([H1|T], [H2|Y]) :-  
    reverse(T, Y).
```

Is program correct? no**Is program recognised as correct?** no**Algorithm used :** unclear**Algorithm detected :** 2 alternative interpretations given – naive & inverse naive
{results of analysis not shown here since algorithm is unclear}

Program version 15 :**Frequency : 1**

```
reverse([], Y).  
reverse([XH|XT], Y) :-  
    reverse(XT, Y),  
    append([Y|XH]).  
append(X).
```

Is program correct? no**Is program recognised as correct?** no**Algorithm used :** unclear**Algorithm detected :** naive*{results of analysis not shown here since algorithm is unclear}*

Program version 16 :**Frequency : 4**

```
reverse([], []).  
reverse([H|T], Res) :-  
    reverse(T, Sofar),  
    append(Sofar, [H], Res).
```

% bug – missing append/3 definition

Is program correct? no**Is program recognised as correct?** no**Algorithm used :** naive**Algorithm detected :** naive**No of bugs in program :** 1 **No of bugs detected and corrected :** 1 **No of bugs detected but corrected wrongly :** 0 **No of bugs not detected :** 0 **No of false alarms :** 0

Program version 17 :**Frequency : 9**

```
reverse([], []).  
reverse([H|T], Res) :-  
    reverse(T, Sofar),  
    append(Sofar, [H], Res).  
append([], L, L).  
append([H|L1], L2, [H|L3]) :-  
    append(L1, L2, L3).
```

Is program correct? yes**Is program recognised as correct? yes****Algorithm used : naive****Algorithm detected : naive****No of bugs in program : 0****No of bugs detected and corrected : 0****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**

Program version 18 :**Frequency : 1**

```
app([], L, L).  
app([H|L1], L2, [H|L3]) :-  
    app(L1, L2, L3).  
reverse([], []).  
reverse([H|T], Res) :-  
    app(Sofar, [H], Res),  
    reverse(T, Sofar).
```

Is program correct? yes**Is program recognised as correct? yes****Algorithm used : naive****Algorithm detected : naive****No of bugs in program : 0****No of bugs detected and corrected : 0****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**

Program version 19 :**Frequency : 1**

```
app1([H|L1], L2, [H|L3]) :-  
    app1(L1, L2, L3).  
app1([], L, L).  
reverse([], []).  
reverse([H|T], Res) :-  
    app1(Sofar, [H], Res),  
    reverse(T, Sofar).
```

Is program correct? no**Is program recognised as correct? yes****Algorithm used : naive****Algorithm detected : naive****No of bugs in program : 1****No of bugs detected and corrected : 0****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 1****No of false alarms : 0**

Program version 20 :**Frequency : 1**

```
reverse([], []).
reverse([H|T], Res) :-
    reverse(T, Sofar),
    append(Sofar, [H], Res).
append([], L, L).
append([H|L1], L2, [H, L3]) :-                % bug – append([H|L1], L2, [H|L3]) :-
    append(L1, L2, L3).
Is program correct? no
Is program recognised as correct? no
Algorithm used : naive
Algorithm detected : naive
No of bugs in program : 1
    No of bugs detected and corrected : 1
    No of bugs detected but corrected wrongly : 0
    No of bugs not detected : 0
    No of false alarms : 0
```

Program version 21 :**Frequency : 4**

```
reverse([], []).
reverse([H|T], Res) :-
    reverse(T, Sofar),
    append(Sofar, H, Res).                    % bug – append(Sofar, [H], Res).
                                           % bug – missing append/3 definition
Is program correct? no
Is program recognised as correct? no
Algorithm used : naive
Algorithm detected : naive
No of bugs in program : 2
    No of bugs detected and corrected : 2
    No of bugs detected but corrected wrongly : 0
    No of bugs not detected : 0
    No of false alarms : 0
```

Comment : This program can also be corrected by leaving the append/3 subgoal as it is, and adding the appropriate append/3 definition.

Program version 22 :**Frequency : 1**

```
reverse([], []).
reverse([XH|XT], Y) :-
    reverse(XT, Y),                          % bug – reverse(XT, X),
    append(XH, XT, Y).                       % bug – append(X, [XH], Y).
append([], L, L).
append([H|L1], L2, [H|L3]) :-
    append(L1, L2, L3).
Is program correct? no
Is program recognised as correct? no
Algorithm used : naive
Algorithm detected : naive
No of bugs in program : 2
    No of bugs detected and corrected : 2
    No of bugs detected but corrected wrongly : 0
    No of bugs not detected : 0
    No of false alarms : 0
```

Program version 23 :**Frequency : 1**

```
reverse([], []).
reverse([XH|XT], Y) :-
    reverse(XT, Y),
    append([XH], XT, Y).
append([], L, L).
append([H|L1], L2, [H|L3]) :-
    append(L1, L2, L3).
```

Is program correct? no
Is program recognised as correct? no
Algorithm used : naive
Algorithm detected : naive
No of bugs in program : 2
 No of bugs detected and corrected : 2
 No of bugs detected but corrected wrongly : 0
 No of bugs not detected : 0
 No of false alarms : 0

Program version 24 :**Frequency : 1**

```
reverse([], []).
reverse([H|T], Res) :-
    reverse(T, Sofar),
    append(Sofar, H, Res).
append([], L, L).
append([H|L1], L2, [H|L3]) :-
    append(L1, L2, L3).
```

Is program correct? no
Is program recognised as correct? no
Algorithm used : naive
Algorithm detected : naive
No of bugs in program : 2
 No of bugs detected and corrected : 2
 No of bugs detected but corrected wrongly : 0
 No of bugs not detected : 0
 No of false alarms : 0

Program version 25 :**Frequency : 2**

```
reverse([], []).
reverse([H|T], Res) :-
    reverse(T, Sofar),
    append(Sofar, H, Res).
append([], L, L).
append([H|L1], L2, [H|L3]) :-
    append(L1, L2, L3).
```

Is program correct? no
Is program recognised as correct? no
Algorithm used : naive
Algorithm detected : naive
No of bugs in program : 1
 No of bugs detected and corrected : 1
 No of bugs detected but corrected wrongly : 0
 No of bugs not detected : 0
 No of false alarms : 0

Program version 26 :**Frequency : 2**

reverse([], []).

reverse([H|T], Res) :-

reverse(T, Sofar),

append(Sofar, H, Res).

% bug – append(Sofar, [H], Res).

append([], L, L).

append([H|L1], L2, [H, L3]) :-

append(L1, L2, L3).

% bug – append([H|L1], L2, [H|L3]) :-

Is program correct? no**Is program recognised as correct? no****Algorithm used : naive****Algorithm detected : naive****No of bugs in program : 2****No of bugs detected and corrected : 2****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**

Program version 27 :**Frequency : 1**

reverse([], []).

reverse([H|T], Res) :-

append(Sofar, [H], Res),

reverse(T, Sofar).

% bug – missing append/3 definition

Is program correct? no**Is program recognised as correct? no****Algorithm used : naive****Algorithm detected : naive****No of bugs in program : 1****No of bugs detected and corrected : 1****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**

Program version 28 :**Frequency : 1**

reverse([], []).

reverse(X, Y) :-

X = [H|T],

append(Y, H, Newlist),

% bug – append(Newlist, H, Y),

reverse(T, Newlist).

% bug – missing append/3 definition

Is program correct? no**Is program recognised as correct? no****Algorithm used : naive****Algorithm detected : naive****No of bugs in program : 2****No of bugs detected and corrected : 2****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**

Program version 29 :**Frequency : 2**

```
reverse([], []).
reverse([H|T], Res) :-
    append(Sofar, [H], Res),
    reverse(T, Sofar).
append([], List, List).
append([H|T], List2, Result) :-
    Result = [H|X],
    append(T, List2, X).
```

Is program correct? yes**Is program recognised as correct? yes****Algorithm used : naive****Algorithm detected : naive****No of bugs in program : 0****No of bugs detected and corrected : 0****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**

Program version 30 :**Frequency : 1**

```
reverse([], []).
reverse([Head|Tail], X) :-
    append(Y, [Head], X),
    reverse(Y, Tail).                % bug - reverse(Tail, Y).
append([], List, List).
append([H|T], List2, Result) :-
    Result = [H|X],
    append(T, List2, X).
```

Is program correct? yes**Is program recognised as correct? no****Algorithm used : naive****Algorithm detected : naive****No of bugs in program : 0****No of bugs detected and corrected : 0****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 1**

Program version 31 :**Frequency : 1**

```
reverse([], []).
reverse([Head|Tail], X) :-
    conc(Y, [Head], X),
    reverse(Y, Tail).                % bug - reverse(Tail, Y).
                                     % bug - missing conc/3 definition
```

Is program correct? no**Is program recognised as correct? no****Algorithm used : naive****Algorithm detected : naive****No of bugs in program : 2****No of bugs detected and corrected : 2****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**

Program version 32 :**Frequency : 1**

reverse([], []).

reverse([H|T], X) :-

append(X, H, Y),

reverse(T, Y).

% bug – append(Y, [H], X)

% bug – missing append/3 definition

Is program correct? no**Is program recognised as correct?** no**Algorithm used :** naive**Algorithm detected :** naive**No of bugs in program :** 2

No of bugs detected and corrected : 2

No of bugs detected but corrected wrongly : 0

No of bugs not detected : 0

No of false alarms : 0

Comment : This program can also be corrected by leaving the append/3 subgoal as it is, and adding the appropriate append/3 definition.

Program version 33 :**Frequency : 1**

reverse([], []).

reverse([H|T], X) :-

add(H, X, Y),

reverse(T, Y).

% bug – add(Y, [H], X),

% bug – missing add/3 definition

Is program correct? no**Is program recognised as correct?** no**Algorithm used :** naive**Algorithm detected :** 2 alternative interpretations – naive & inverse naive**No of bugs in program :** 2

No of bugs detected and corrected : 2

No of bugs detected but corrected wrongly : 0

No of bugs not detected : 0

No of false alarms : 0

Comment : This program can also be corrected by leaving the add/3 subgoal as it is, and adding the appropriate add/3 definition.

Program version 34 :**Frequency : 1**

reverse([], []).

reverse([H|T], Y) :-

append(H, Y, Y),

reverse(T, Y).

% bug – append(X, [H], Y),

% bug – reverse(T, X).

% bug – missing append/3 definition

Is program correct? no**Is program recognised as correct?** no**Algorithm used :** naive**Algorithm detected :** naive**No of bugs in program :** 3

No of bugs detected and corrected : 3

No of bugs detected but corrected wrongly : 0

No of bugs not detected : 0

No of false alarms : 0

Program version 35 :**Frequency : 1**

reverse([], []).

reverse([H|T], Res) :-

append(Sofar, H, Res),

reverse(T, Sofar).

% bug – append(Sofar, [H], Res)

% bug – missing append/3 definition

Is program correct? no

Is program recognised as correct? no

Algorithm used : naive

Algorithm detected : naive

No of bugs in program : 2

No of bugs detected and corrected : 2

No of bugs detected but corrected wrongly : 0

No of bugs not detected : 0

No of false alarms : 0

Comment : This program can also be corrected by leaving the append/3 subgoal as it is, and adding the appropriate append/3 definition.

Program version 36 :**Frequency : 4**

reverse([], []).

reverse([H|T1], [T2|H]) :-

reverse(T1, T2).

% bug – reverse([H|T1], X) :-

% bug – missing append(T2, [H], X) subgoal

% bug – missing append/3 definition

Is program correct? no

Is program recognised as correct? no

Algorithm used : naive

Algorithm detected : naive

No of bugs in program : 3

No of bugs detected and corrected : 3

No of bugs detected but corrected wrongly : 0

No of bugs not detected : 0

No of false alarms : 0

Program version 37 :**Frequency : 1**

reverse([], Y).

reverse([H|T1], [T2|H]) :-

reverse(T1, T2).

% bug – reverse([], []).

% bug – reverse([H|T1], X) :-

% bug – missing append(T2, [H], X) subgoal

% bug – missing append/3 definition

Is program correct? no

Is program recognised as correct? no

Algorithm used : naive

Algorithm detected : naive

No of bugs in program : 4

No of bugs detected and corrected : 4

No of bugs detected but corrected wrongly : 0

No of bugs not detected : 0

No of false alarms : 0

Program version 38 :**Frequency : 1**

```
reverse([], []).  
reverse([N], [N]).  
reverse([A|B], [C|A]) :-  
    reverse(B, C).  
% bug - extra clause  
% bug - reverse([A|B], X) :-  
  
% bug - missing append(C, [A], X) subgoal  
% bug - missing append/3 definition
```

Is program correct? no**Is program recognised as correct? no****Algorithm used : naive****Algorithm detected : naive****No of bugs in program : 4****No of bugs detected and corrected : 4****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**

Program version 39 :**Frequency : 1**

```
reverse([], []).  
reverse([X], [X]).  
reverse([Head|Tail], [Contents_of_reverse_tail|Head]) :-  
    reverse(Tail, [Contents_of_reverse_tail]).  
% bug - extra clause  
% bug - reverse([Head|Tail], X) :-  
  
% bug - reverse(Tail, Y),  
% bug - missing append(Y, [Head], X) subgoal  
% bug - missing append/3 definition
```

Is program correct? no**Is program recognised as correct? no****Algorithm used : naive****Algorithm detected : naive****No of bugs in program : 5****No of bugs detected and corrected : 5****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**

Program version 40 :**Frequency : 1**

```
reverse([], []).  
reverse([Head|Tail], [X|Head]) :-  
    reverse(X, Tail).  
% bug - reverse([], X, X).  
% bug - reverse([Head|Tail], X, Y) :-  
% bug - reverse(X, [Head|Tail], Y).  
% bug - missing clause :  
%  
% reverse(X, Y) :-  
% reverse(X, [], Y).
```

Is program correct? no**Is program recognised as correct? no****Algorithm used : naive****Algorithm detected : railway-shunt****No of bugs in program : 4****No of bugs detected and corrected : 4****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**

Program version 41 :**Frequency : 1**

```
append([], L, L).
append([H], L, [H|L]).
append([H|T], L, Ans) :-
    append(T, L, X),
    append([H], X, Ans).
reverse([], []).
reverse([H], [H]).
reverse([H|T], L) :-
    reverse(T, M),
    append(M, [H|L], L). % bug – append (M, [H], L).
```

Is program correct? no**Is program recognised as correct? no****Algorithm used : naive****Algorithm detected : naive****No of bugs in program : 1****No of bugs detected and corrected : 1****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**

Program version 42 :**Frequency : 1**

```
append([], L, L).
append([H], L, [H|L]).
append([H|T], L, Ans) :-
    append(T, L, X),
    append([H], X, Ans).
reverse([], []).
reverse([H], [H]).
reverse([H|T], L) :-
    reverse(T, M),
    append(M, H, L). % bug – append (M, [H], L).
```

Is program correct? no**Is program recognised as correct? no****Algorithm used : naive****Algorithm detected : naive****No of bugs in program : 1****No of bugs detected and corrected : 1****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**

Program version 43 :**Frequency : 1**

```
append([], L, L).
append([H], L, [H|L]).
append([H|T], L, Ans) :-
    append(T, L, X),
    append([H], X, Ans).
reverse([], []).
reverse([H], [H]).
reverse([H|T], L) :-
    reverse(T, M),
    append(M, [H], L).
```

Is program correct? yes**Is program recognised as correct? yes****Algorithm used : naive****Algorithm detected : naive****No of bugs in program : 0****No of bugs detected and corrected : 0****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**

Comment : The append/3 predicate definition produces infinite identical solutions for append(+, +, -).

Program version 44 :**Frequency : 1**

```
reverse([], L).                                % bug - reverse([], []).
reverse([H], Ans) :-                            % bug - extra clause
    Ans = [Ans|H]).
reverse([H|T], Ans) :-
    Ans = [Ans|H],                             % bug - extra subgoal
    reverse(T, X).                             % bug - reverse(T, X).
                                                % bug - missing subgoal
                                                %          "append(X, [H], Ans)."
                                                % bug - missing append/3 definition
```

Is program correct? no**Is program recognised as correct? no****Algorithm used : naive****Algorithm detected : naive****No of bugs in program : 6****No of bugs detected and corrected : 6****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**

Program version 45 :**Frequency : 4**

```
reverse([], Y).  
reverse([H|T], Y) :-  
    reverse(T, [H|Y]).
```

```
% bug - reverse([], Y, Y).  
% bug - reverse([H|T], Y, X) :-  
% bug - reverse(T, [H|Y], X).  
% missing clause :  
%     reverse(X, Y) :-  
%         reverse(X, [], Y).
```

Is program correct? no**Is program recognised as correct? no****Algorithm used : railway-shunt****Algorithm detected : railway-shunt****No of bugs in program : 4****No of bugs detected and corrected : 4****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**

Program version 46 :**Frequency : 1**

```
reverse([], Y).  
reverse([H|T], Y) :-  
    reverse(T, [H|_]).
```

```
% bug - reverse([], []).  
  
% bug - reverse(T, X),  
% bug - missing subgoal "append(X, [H], Y)."  
% bug - missing append/3 definition
```

Is program correct? no**Is program recognised as correct? no****Algorithm used : railway-shunt****Algorithm detected : naive****No of bugs in program : 4****No of bugs detected and corrected : 4****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**

Program version 47 :**Frequency : 9**

```
reverse([], []).  
reverse([H|T], Y) :-  
    reverse(T, [H|Y]).
```

```
% bug - reverse(T, X),  
% bug - missing subgoal "append(X, [H], Y)."  
% bug - missing append/3 definition
```

Is program correct? no**Is program recognised as correct? no****Algorithm used : railway-shunt****Algorithm detected : naive****No of bugs in program : 3****No of bugs detected and corrected : 3****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**

Program version 48 :**Frequency : 1**

reverse([], []).

reverse(X, Y) :-

X = [H|T],

reverse(T, [H|Y]).

% bug - reverse(T, X),

% bug - missing subgoal "append(X, [H], Y)."

% bug - missing append/3 definition

Is program correct? no**Is program recognised as correct? no****Algorithm used : railway-shunt****Algorithm detected : naive****No of bugs in program : 3****No of bugs detected and corrected : 3****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**

Program version 49 :**Frequency : 1**

reverse([], []).

reverse(X, Y) :-

X = [H|T],

New = [H|Y],

reverse(T, New).

% bug - reverse(T, X),

% bug - missing subgoal "append(X, [H], Y)."

% bug - missing append/3 definition

Is program correct? no**Is program recognised as correct? no****Algorithm used : railway-shunt****Algorithm detected : naive****No of bugs in program : 3****No of bugs detected and corrected : 3****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**

Program version 50 :**Frequency : 2**

reverse([], []).

reverse([], Y).

reverse([H|T], [X]) :-

reverse(T, [X|H]).

% bug - reverse([], Y, Y).

% bug - reverse([H|T], X, Y) :-

% bug - reverse(T, [H|X], Y).

% missing clause :

% reverse(X, Y) :-

% reverse(X, [], Y).

Is program correct? no**Is program recognised as correct? no****Algorithm used : railway-shunt****Algorithm detected : railway-shunt****No of bugs in program : 4****No of bugs detected and corrected : 4****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**

Program version 51 :**Frequency : 1**

```
reverse([], []).
reverse([], Y).
reverse([H|T], Y) :-
    reverse(T, [H|Y]).
reverse(Y, []).
```

```
% bug - reverse([], Y, Y).
% bug - reverse([H|T], X, Y) :-
% bug - reverse(T, [H|X], Y).
% bug - extra clause
% missing clause :
%     reverse(X, Y) :-
%         reverse(X, [], Y).
```

Is program correct? no**Is program recognised as correct?** no**Algorithm used :** railway-shunt**Algorithm detected :** railway-shunt**No of bugs in program :** 5**No of bugs detected and corrected :** 5**No of bugs detected but corrected wrongly :** 0**No of bugs not detected :** 0**No of false alarms :** 0

Program version 52 :**Frequency : 1**

```
reverse([], []).
reverse([], Y).
reverse([H|T], [X]) :-
    reverse(T, [H|X]).
```

```
% bug - reverse([], Y, Y).
% bug - reverse([H|T], X, Y) :-
% bug - reverse(T, [H|X], Y).
% missing clause :
%     reverse(X, Y) :-
%         reverse(X, [], Y).
```

Is program correct? no**Is program recognised as correct?** no**Algorithm used :** railway-shunt**Algorithm detected :** railway-shunt**No of bugs in program :** 4**No of bugs detected and corrected :** 4**No of bugs detected but corrected wrongly :** 0**No of bugs not detected :** 0**No of false alarms :** 0

Program version 53 :**Frequency : 1**

```
reverse([], Y).
reverse([], []).
reverse([H|T], Y) :-
    reverse(T, [H|Y]).
```

```
% bug - reverse([], Y, Y).
% bug - reverse([H|T], X, Y) :-
% bug - reverse(T, [H|X], Y).
% missing clause :
%     reverse(X, Y) :-
%         reverse(X, [], Y).
```

Is program correct? no**Is program recognised as correct?** no**Algorithm used :** railway-shunt**Algorithm detected :** railway-shunt**No of bugs in program :** 4**No of bugs detected and corrected :** 4**No of bugs detected but corrected wrongly :** 0**No of bugs not detected :** 0**No of false alarms :** 0

Program version 54 :**Frequency : 1**

```
reverse([], Y).                                % extra clause
reverse([], []).
reverse(A, B) :-
    recur(A, B).                                % bug - recur(A, [], B).
recur([H|T], Y) :-                             % bug - recur([H|T], Y, X) :-
    recur(T, [H|Y]).                          % bug - recur(T, [H|Y], X).
                                              % missing clause :
                                              %      recur([], X, X).
```

Is program correct? no**Is program recognised as correct? no****Algorithm used : railway-shunt****Algorithm detected : railway-shunt****No of bugs in program : 5****No of bugs detected and corrected : 5****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**

Program version 55 :**Frequency : 1**

```
reverse([H|T], Y) :-                          % bug - reverse([H|T], X, Y) :-
    reverse(T, [H|Y]).                        % bug - reverse(T, [H|X], Y).
                                              % missing clauses :
                                              %      reverse([], X, X).
                                              %      reverse(X, Y) :-
                                              %          reverse(X, [], Y).
```

Is program correct? no**Is program recognised as correct? no****Algorithm used : railway-shunt****Algorithm detected : railway-shunt****No of bugs in program : 4****No of bugs detected and corrected : 4****No of bugs detected but corrected wrongly : 0****No of bugs not detected : 0****No of false alarms : 0**
